

FILE CODE

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California

2

AD-A214 471



# THESIS

THE REAL-TIME ROLL-BACK AND RECOVERY OF  
TRANSACTIONS IN DATABASE SYSTEMS

by

David E. Quantock

June 1989

Thesis Advisor:

David K. Hsiao

Approved for Public Release; Distribution is Unlimited

DTIC  
ELECTE  
NOV 22 1989  
S B D

89 11 20 103

# REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>		1b RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION AVAILABILITY OF REPORT <b>Approved for Public Release; Distribution is Unlimited</b>	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE			
4 PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a NAME OF PERFORMING ORGANIZATION <b>Naval Postgraduate School</b>	6b OFFICE SYMBOL (If applicable) <b>Code 52</b>	7a NAME OF MONITORING ORGANIZATION <b>Naval Postgraduate School</b>	
6c ADDRESS (City, State, and ZIP Code) <b>Monterey, California 93943-5000</b>		7b ADDRESS (City, State, and ZIP Code) <b>Monterey, California 93943-5000</b>	
8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO	PROJECT NO
		TASK NO	WORK UNIT ACCESSION NO
11 TITLE (Include Security Classification) <b>THE REAL-TIME ROLL-BACK AND RECOVERY OF TRANSACTIONS IN DATABASE SYSTEMS</b>			
12 PERSONAL AUTHOR <b>Quantock, David E.</b>			
13a TYPE OF REPORT <b>Master's Thesis</b>	13b TIME COVERED FROM TO	14 DATE OF REPORT (Year, Month, Day) <b>1989, June</b>	15 PAGE COUNT <b>94</b>
16 SUPPLEMENTARY NOTES			
17		18 SUBJECT TERMS (Continue on reverse if necessary; and identify by block number)	
FIELD GROUP		Roll-back and Recovery, Multilingual Database System, Multibackend Database System, Incremental Logging, Differential Files, Shadow Paging, Backend Transaction Log.	
19 ABSTRACT (Continue on reverse if necessary; and identify by block number)			
<p>A modern database transaction may involve a long series of updates, deletions, and insertions of data and a complex mix of these primary database operations. Due to its length and complexity, the transaction requires back-up and recovery procedures. The back-up procedure allows the user to either commit or abort a lengthy and complex transaction without compromising the integrity of the data. The recovery procedure allows the system to maintain the data integrity during the execution of a transaction, should the transaction be interrupted by the system.</p> <p>With both the back-up and recovery procedures, the modern database system will be able to provide consistent data throughout the life-span of a database without ever corrupting either its data values or its data types.</p> <p>However, the implementation of back-up and recovery procedures in a database system is</p>			
20 DISTRIBUTION STATEMENT (See instructions for use)		21 ABSTRACT SECURITY CLASSIFICATION	
<input checked="" type="checkbox"/> UNCLASSIFIED		<b>Unclassified</b>	
22 NAME OF AUTHOR		23 TELEPHONE (Include Area Code) and OFFICE SYMBOL	
<b>Prof. David K. Esiao</b>		<b>(408) 646-2253 Code 52Bc</b>	

DD FORM 1475, 1-89

UNCLASSIFIED

## #19 - ABSTRACT - (CONTINUED)

a difficult and involved effort since it effects the base as well as meta data of the database. Further, it effects the state of the database system. This thesis is mainly focused on the design trade-offs and issues of implementing an effective and efficient mechanism for back-up and recovery in the multimodel, multilingual, and multibackend database system.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for public release; distribution is unlimited

The Real-Time Roll-Back and Recovery of Transactions in Database  
Systems

by

David E. Quantock  
Captain, United States Army  
B.A., Norwich University, 1980  
M.S., Troy State University, 1983

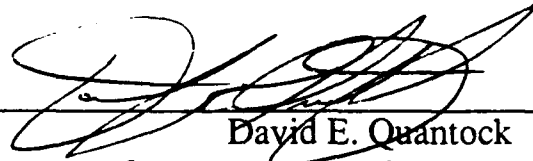
Submitted in partial fulfillment of the  
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE


from the

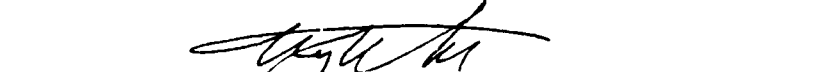
NAVAL POSTGRADUATE SCHOOL  
June 1989


Author:


  
David E. Quantock

Approved by:

  
David K. Hsiao, Professor of Computer Science  
Thesis Advisor

  
Thomas Wu, Associate Professor of Computer Science  
Second Reader

  
Robert McGhee, Chairman  
Department of Computer Science

  
Kneale T. Marshall  
Dean of Information and Policy Sciences

## ABSTRACT

A modern database transaction may involve a long series of updates, deletions, and insertions of data and a complex mix of these primary database operations. Due to its length and complexity, the transaction requires back-up and recovery procedures. The back-up procedure allows the user to either commit or abort a lengthy and complex transaction without compromising the integrity of the data. The recovery procedure allows the system to maintain the data integrity during the execution of a transaction, should the transaction be interrupted by the system.

With both the back-up and recovery procedures, the modern database system will be able to provide consistent data throughout the life-span of a database without ever corrupting either its data values or its data types.

However, the implementation of back-up and recovery procedures in a database system is a difficult and involved effort since it effects the base as well as meta data of the database. Further, it effects the state of the database system. This thesis is mainly focused on the design trade-offs and issues of implementing an effective and efficient mechanism for back-up and recovery in the multimodel, multilingual, and multibackend database system.

## TABLE OF CONTENTS

<b>I. INTRODUCTION .....</b>	<b>1</b>
<b>A. THE MOTIVATION .....</b>	<b>1</b>
1. Roll-back and Recovery Issues .....	2
2. Design Decisions .....	3
<b>B. THE SYSTEM BACKGROUND .....</b>	<b>4</b>
1. The Multilingual Database System .....	5
2. The Multibackend Database System .....	9
<b>C. THESIS ORGANIZATION .....</b>	<b>12</b>
<b>II. ROLL-BACK ALGORITHM OPTIONS .....</b>	<b>13</b>
<b>A. OVERVIEW OF THE THREE GENERAL ALGORITHMS .....</b>	<b>13</b>
<b>B. MODELS OF THREE GENERAL ALGORITHMS .....</b>	<b>13</b>
1. Incremental Logging .....	14
a. Algorithm Description .....	14
b. Advantages .....	16
c. Disadvantages .....	17
2. Differential Files .....	17
a. Algorithm Description .....	18
b. Advantages .....	20
c. Disadvantages .....	21

3. Shadow Paging .....	21
a. Algorithm Description .....	21
b. Advantages .....	22
c. Disadvantages .....	22
C. SUMMARY AND CONCLUSIONS .....	25
III. IMPLEMENTATION ISSUES .....	29
A. IMPLEMENTATION CONSIDERATIONS .....	29
1. Problem Specification .....	29
2. Modifications to the Differential Log .....	29
3. Placement and Description of the Log .....	31
B. THE CENTRALIZED APPROACH .....	31
1. User Interface .....	33
2. General Algorithm .....	35
3. Advantages .....	38
4. Disadvantages .....	38
C. THE DECENTRALIZED APPROACH .....	38
1. User Interface .....	39
2. General Algorithm .....	41
3. Advantages .....	44
4. Disadvantages .....	45
D. SUMMARY AND CONCLUSIONS .....	45

<b>IV. THE BACKEND TRANSACTION LOG (BTL)</b>	<b>48</b>
1. The Data Structure	48
2. Decomposition View of the Algorithm	50
3. The Algorithm	53
4. Summary	62
<b>V. CONCLUSIONS</b>	<b>64</b>
<b>APPENDIX</b>	<b>68</b>
<b>LIST OF REFERENCES</b>	<b>82</b>
<b>BIBLIOGRAPHY</b>	<b>84</b>
<b>INITIAL DISTRIBUTION LIST</b>	<b>85</b>



## LIST OF FIGURES

1.	The Multilingual Database System (MLDS) .....	6
2.	Multiple Language Interfaces for the Same KDS .....	8
3.	The MBDS Process Structure .....	11
4.	Incremental Logging on MBDS Hardware Organization .....	15
5.	Differential File on MBDS Hardware Organization .....	19
6.	Shadow and Current Page Tables .....	24
7.	Backend Transaction Log .....	32
8.	Centralized BTL User Interface .....	34
9.	Centralized Backend Transaction Log (BTL) .....	36
10.	Decentralized BTL User Interface .....	40
11.	Decentralized Backend Transaction Log (BTL) .....	42
12.	The Data Structure of the Backend Transaction Log .....	49
13.	A Decomposition Diagram .....	51
14.	SEARCH_FOR_PTRARRAY .....	54
15.	The BTL Initialization .....	56
16.	Sending RETRIEVEs to Directory Management .....	57
17.	Update the RETRIEVE_REQUEST .....	60
18.	Transaction Inserted into the Database .....	61

## I. INTRODUCTION

### A. THE MOTIVATION

*Roll-back and recovery* is the return of a database system to its previous state after a user error, application error, partial system failure, or supervisor request. Not only must the system return to a previous state, but the database must return to a *consistent* and *correct* state. In order to return to a consistent state, the Database Management System (DBMS) must be able to keep track of ongoing transactions and abort and roll-back uncommitted transactions if the system is interrupted.

*Roll-back and recovery* is a very important component of a database system. Even with its great importance, few articles have been written on this subject as compared to articles on data models, schema design, query languages, access paths, and locking and concurrency control. It would seem that the main reason for this is that system errors are negligible when compared to the total throughput of a computer system. But although there are few system errors, the overhead incurred by a roll-back and recovery algorithm can have a major impact on the system's overall performance. In addition, the inconsistent and incorrectness of the database and states of DBMS due to the lack of roll-back and recovery may be incalculable.

## 1. Roll-back and Recovery Issues

Over the past decade the area of roll-back and recovery has matured as database systems have become increasingly complex. In a single-user system, transactions are put into the system and committed in a serial fashion. If there is a system failure, the user will have to reenter the last transaction if it had not been committed. There is no single-user system available that could give the user the ability for *supervisor roll-back*. By *supervisor roll-back* we meant that the supervisor could put in a test transaction and see the results and then undo the results of the transaction if desired.

Now in the era of distributed computer databases, you have databases distributed over a network with multiple users trying to access the data. This requires a much more complex DBMS to handle the many user transactions and then roll-back to a consistent database in the event of a system interrupt.

Articles written on roll-back and recovery all seem to concentrate on roll-back after a system failure [Refs. 1,2,3,4,5,6,7,8,9,10,11]. In 1981 a study was done by IBM on the performance of their first relational database system, System R [Ref. 3:p. 226]. One of their findings was that 97% of all transactions execute successfully. In the remaining 3% of the transactions, almost all result from incorrect user input. Less than 1% of the transactions are aborted because of system failure (i.e., system overload or deadlock). It

was these findings that generated the feeling that roll-back and recovery may have other applications other than recovery from system error.

A very powerful extension of roll-back and recovery would be the ability to insert a transaction, allow the supervisor to see the results of that transaction, and then if he/she liked the results, have the transaction committed to the database. If the results were unsatisfactory to the supervisor, then the supervisor could invoke what we call a *supervisor roll-back*. *Supervisor roll-back* would restore the system to its previous database state. This would allow the supervisor to do a number of tests without committing the database to the transactions. This would be an extremely valuable management tool for testing proposed changes to a database.

## 2. Design Decisions

In designing a roll-back and recovery algorithm there is a number key factors that must be taken into consideration. First, it incurs modest *storage overhead* for the duplicated data that is brought into the main memory. For example, a system that duplicates the entire database for each transaction will quickly run out of storage. Second, the type of *data structure* that stores the data should hold only the data it needs and should dynamically grow and shrink to fit the system needs. Third, the cost of the *storage overhead* versus the number of *rollbacks required*. A system that seldom has a need to roll-back should have a different algorithm than a system that frequently rolls-

back. Fourth, it should *permit parallelism* to the maximum extent possible to satisfy system performance requirements. A system could always guarantee a consistent view of the database and fast recovery if it serially executed each transaction. But the costs of serial execution negate any performance benefit of a concurrent system. Additionally, the level of granularity must be to the level that offers maximum concurrency. For example, the granularity of page logging is generally more costly than entry logging. The reason for this is simply that in page logging the entire contents of the page is logged as opposed to entry logging where only the specific record or modified record is logged. The cost is in terms of increased storage requirements and execution time. Furthermore, page logging implies page locking which impedes concurrency. [Ref. 7:p. 556] Fifth, it should perform satisfactorily in a *network environment* that has some measure of communication delay. Sixth, the overhead during *normal performance* should not be degraded by the roll-back and recovery mechanism. Seventh, *recovery speed* during roll-back should not cause major delays to the users. Eighth, *software complexity* of the recovery mechanism needs to be as simple as possible to prevent system delays. And finally, the recovery system must be *reliable*.

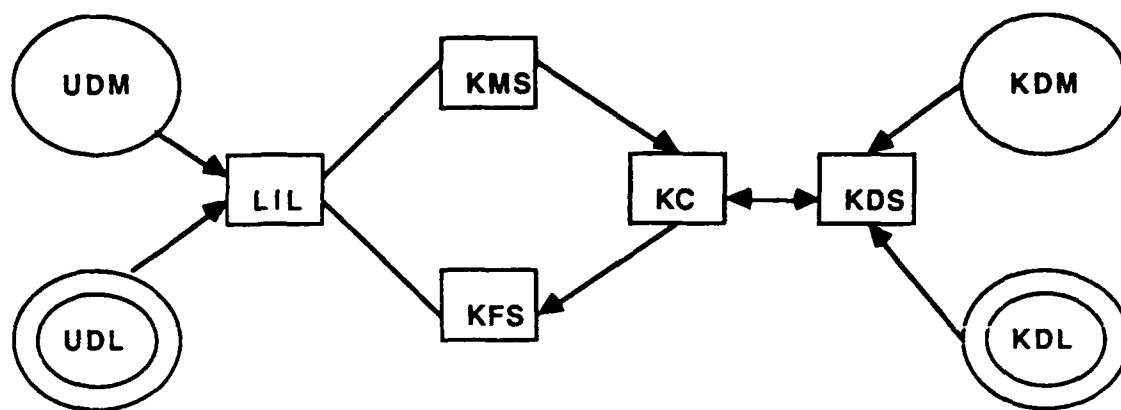
## B. THE SYSTEM BACKGROUND

The vehicle for studying this roll-back and recovery algorithm is the multilingual database system (MLDS) developed at the Naval Postgraduate

School and Ohio State University [Ref. 12,13,14,15]. In this section, we give the reader some background material on the system structure and functions. This will also include an introduction into the architecture of the multibackend database system (MBDS) used to support MLDS database transaction processing.

### **1. The Multilingual Database System**

The multilingual database system is depicted in Figure 1. In this figure, a query, which is written in the user's data language (UDL) (e.g., SQL), is sent into the system and is based on the user's data model (UDM)(e.g., the relational data model). The user's transaction is processed through the language interface layer (LIL) which routes the transaction to the kernel mapping system (KMS). KMS performs two functions. If the user is creating a new database, then the first function for KMS is to transform the database in UDM into an equivalent database in the kernel data model (KDM). In other words, a schema is made up for the transformation of the database in the user model into one in the system's model. KMS then sends the new schema to the kernel controller subsystem (KCS) which in turn sends the KDM-database definition to the kernel database system (KDS). After KDS has received the new KDM-based schema, it notifies KCS who notifies the user that the database may now be loaded.



UDM :User Data Model  
 UDL :User Data Language  
 LIL :Language Interface Layer  
 KMS :Kernel Mapping System  
 KC :Kernel Controller  
 KFS :Kernel Formatting System  
 KDM :Kernel Data Model  
 KDL :Kernel Data Language  
 KDS :Kernel Database System





 Data Language  
 Data Model  
 System Module  
 Information Flow

Figure 1. The MultiLingual Database System (MLDS)

After the schema of the database has been laid out, the second task of KMS is to handle transactions from the user. When a user sends in his query, the UDL (e.g., SQL query), KMS translates the query into the an equivalent KDL version. KMS then sends the KDL transaction to KDS for execution. KDS retrieves or executes the transaction and then sends the results back to KCS in the KDM form. KCS then forwards these results back to the kernel formatting system (KFS) which transforms the results from the format in KDM to the UDM. Once the transformation has completed, KFS sends the results to the user through LIL.

The LIL, KMS, KCS, and KFS components are referred to as the *language interface*. For each user-defined language(model), there must be an interface that takes the user's query(database) and translates(transforms) it into the kernel language(model-based) query(database). For example, there exists a set of language interfaces - one for the relational database/SQL language, another for the hierarchical database/DL/1 language, a third for the network database/CODASYL-DML language and a fourth for the functional database/Daplex language as depicted in Figure 2. All of these interfaces are supported on a single KDS which accesses, stores, and retrieves the data from the databases.

KDM and KDL of the MBDS are the attribute-based data model (ABDM) and the attribute-based data language (ABDL), respectively. ABDL supports the five primary database operations, INSERT, DELETE,



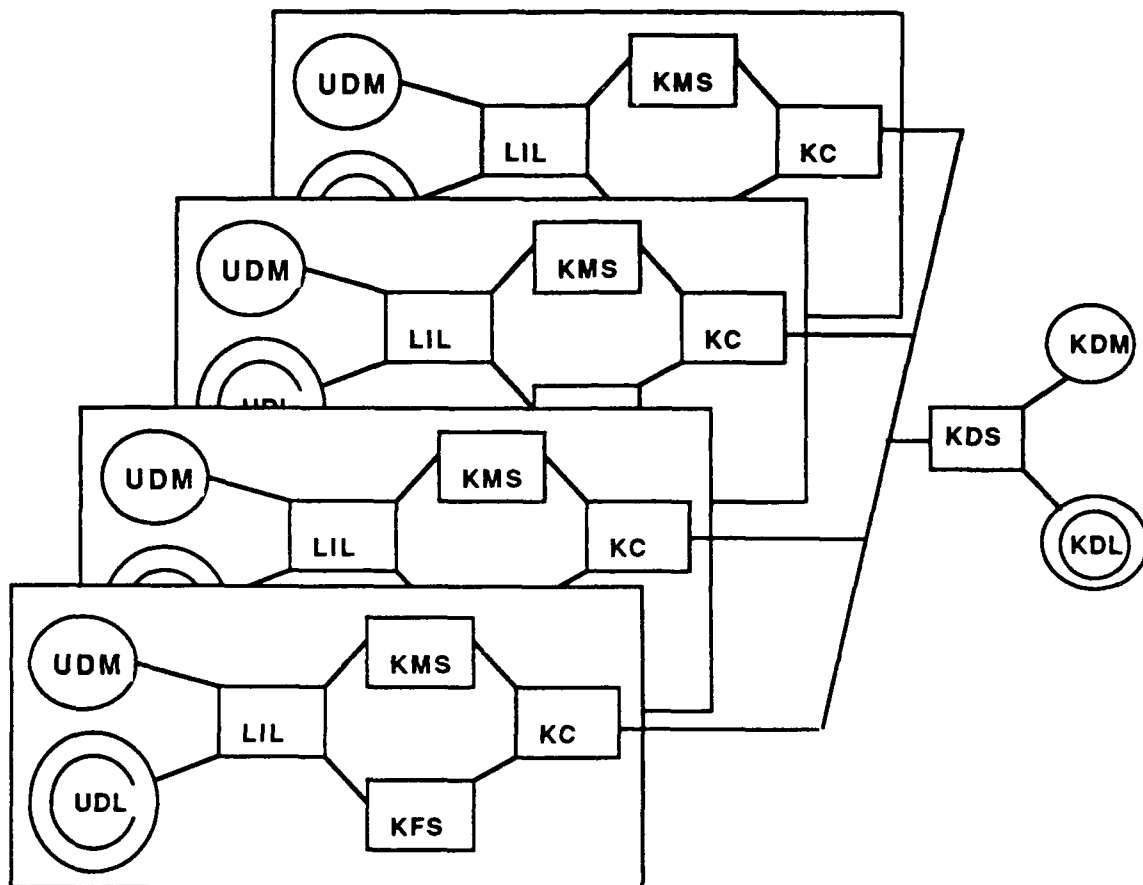


Figure 2. Multiple Language Interfaces for the Same KDS

UPDATE, RETRIEVE, and RETRIEVE-COMMON. The first four operations are obvious. The last operation, RETRIEVE-COMMON, is equivalent to a relational equijoin which provides the merging of two files with common attribute values. These five simple database operations are capable of supporting complex and comprehensive transactions. For a more detailed discussion of the multibackend database system refer to [Ref. 16].

## **2. The Multibackend Database System**

As each year passes, new systems are developed which increase the speed and performance of older systems. This is perplexing for the users who in order to keep up with technology progresses and performance gains, must continually upgrade their software and hardware. While the multilingual database system gives the user the flexibility to incorporate different software interfaces into a system, the concept of the multi-backend database system (MBDS) gives speed and performance upgrades without the conventional upgrade costs in hardware.

By using multiple backends configured in a parallel fashion, performance gains are attained by increasing the number of backends to the system. MBDS will produce nearly a reciprocal decrease in the response times for user transactions when the number of backends is increased. Additionally, if the size of the database increases proportionally with the number of backends added, there will be little if any degradation in system performance [Ref. 17].

One of the major design goals of MBDS was to develop a system that maximized the work of the backends and minimize the work of the controller. In Figure 3, the top half of the diagram depicts the controller and the bottom half the backend. The controller can communicate with 1 or more backends over a local-area network. The controller has three (3) primary functions. First, the controller prepares a request for execution by the backends. This request is performed by the *request preparation* of the controller. Secondly, the controller coordinates responses from the backends. This function is performed by the *post processing*. And lastly, the *insert information generator* (IIG) maintains a status on the current storage capacity of each backend. When an insert request is generated, IIG sends the insert to a specific backend based on the current status of database storage.

The backend also has three but different primary responsibilities: directory management, concurrency control, and record processing. First, the *directory management* receives incoming messages from the controller. The directory manager determines the addresses of the records required to process a particular request. Secondly, the *concurrency control* allows concurrent accesses to the database by different requests. And lastly, the *record processing* section performs data retrieval, storage, and the processing required on any particular record.

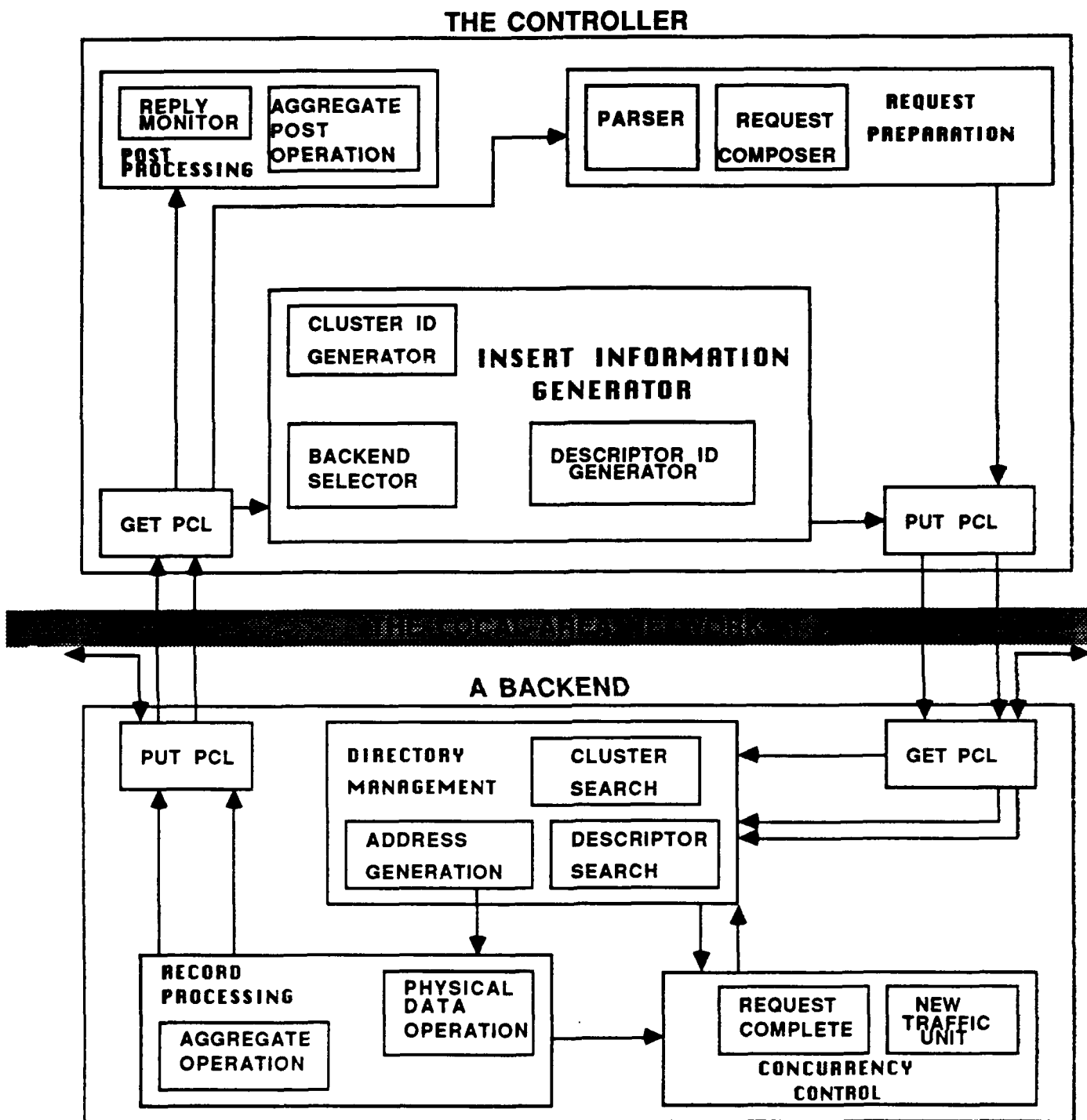


Figure 3. The MBDS Process Structure

For communications over the local-area network, there is a pair of communications processes in each backend and the controller. These processes are called *get pcl* and *put pcl*.

### C. THESIS ORGANIZATION

In this thesis, the three general algorithms for roll-back and recovery were examined: *incremental logging*, *differential files*, and *shadow paging*. With each algorithm, there are many variations. In Chapter II, we provide a description of the general algorithms.

In Chapter III, we introduce a general solution and then present two additional methods to implement the solution. In Chapter IV, we arrive at the recommended solution for implementation in MBDS by giving a description of how the algorithm will work. The actual algorithm, written in pseudo PASCAL, is in the Appendix. Finally, in Chapter V, we make our conclusions about the proposed design.

## II. ROLL-BACK ALGORITHM OPTIONS

### A. OVERVIEW OF THE THREE GENERAL ALGORITHMS

There are generally three basic algorithms in roll-back and recovery. As in any area of study there are a number of derivations from these general solutions. In this Chapter, we describe the three basic algorithms along with the advantages and disadvantages of each approach.

The first general algorithm is *incremental logging*. In incremental logging the technique is to apply the updates directly to the database and keep an incremental log of all changes to the system state. In the second general algorithm, *differential files*, the algorithm calls for a centralized log which defers all updates until the end of the transaction. And finally, *shadow paging* is an alternative to log-based recovery techniques. In shadow paging, the database is partitioned into pages. During a transaction a page has a current page table and the shadow page table. All updates are made to the current page table with the shadow page table as the recovery source in case the transaction fails. These three methods will be discussed in much greater detail in the sections that follow.

### B. MODELS OF THREE GENERAL ALGORITHMS

It should be noted that a *transaction* is decomposed into a number of *requests* that are executed to perform the transaction. Each request is treated as an atomic item. As each request of a transaction completes, the

transaction remains in a *partially committed* state until all requests of the transaction have been completed.

### **1. Incremental Logging**

As stated earlier, the incremental log is a centralized file that logs in transactions. As the transaction is being logged in, it is also being instantaneously entered into the database. If the system fails or has to roll-back, then the system must *undo* the transaction based on the information in the incremental log.

#### ***a. Algorithm Description***

Figure 4 depicts the placement of the incremental log if it were placed in MBDS. As each transaction enters the controller, the system would immediately modify the database or retrieve the data (arrow number 1, Figure 4). In addition, if a write is involved, the system annotates in the incremental log the change to the database (arrow number 2, Figure 4).

Specifically, when a transaction enters the system, a "start" record is written to the log with the transaction number attached. During the execution of the transaction, if write operations are encountered they are entered into the log and simultaneously sent directly to the database for updating. As a minimum, a record consists of a transaction name, data item name, old value of the data, and new value of the data. When the transaction partially commits, a "commit" record is written into the log. As the transaction continues, i.e., the next request is executed, another partially

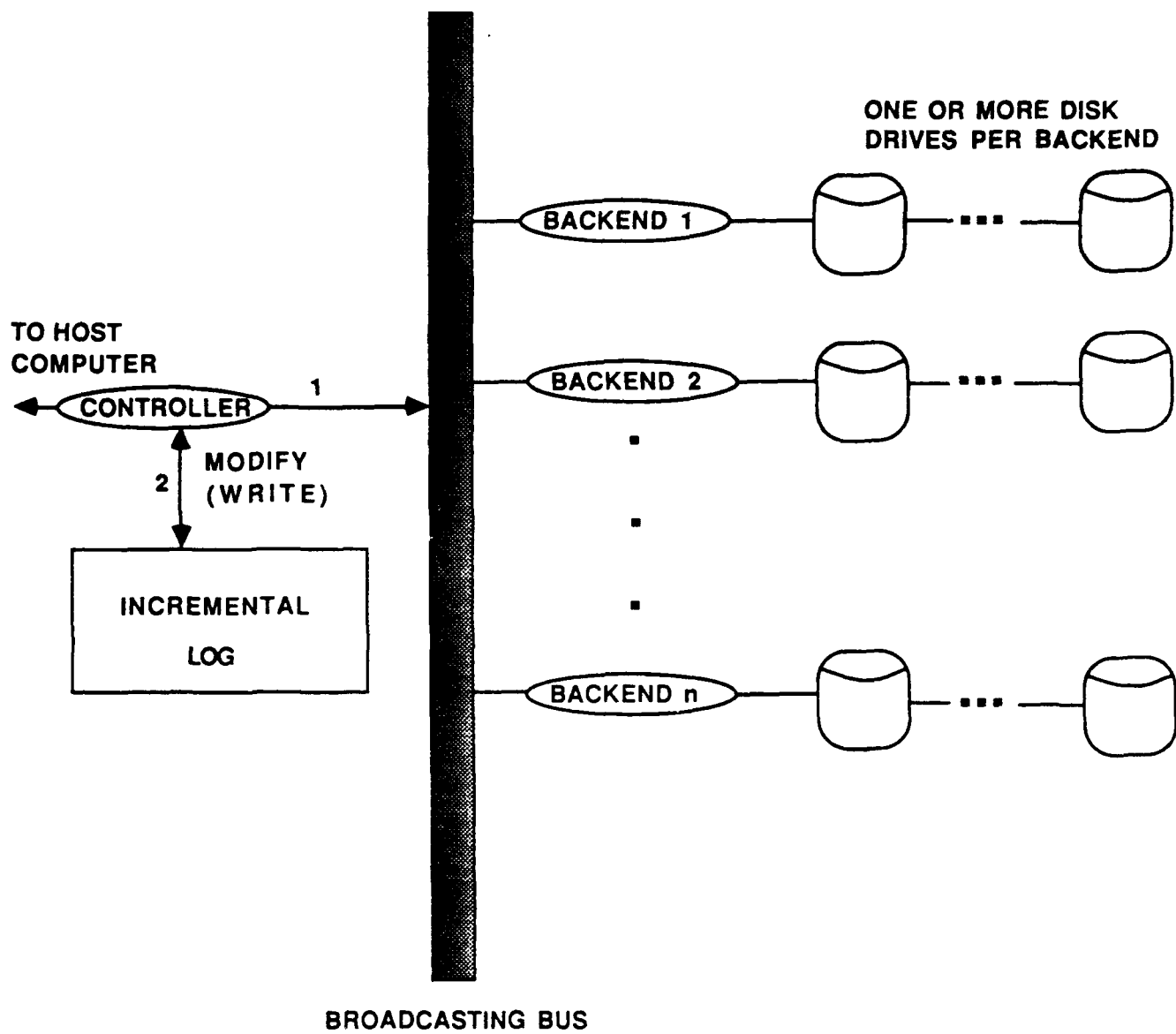


Figure 4. Incremental Logging on MBDS Hardware Organization



committed state is reached. If the system fails in the middle of a request, then the request that had not completed is *undone* and the rest of the transaction that had partially committed is first *undone* and then *redone*. The system is able to accomplish the *undo* and *redo* by referring to the incremental log.

*Checkpointing* can also be used to reduce the overhead of searching the log and redoing transactions. checkpointing is done for primarily two reasons. First, the searching process of the incremental log is time-consuming. Secondly, most of the transactions that need to be redone have already made changes to the database and do not need to be redone. With checkpointing, after a failure has occurred, the system only has to check the log to its last transaction that started executing before the last checkpoint. That transaction is undone and then redone. For more information on checkpointing refer to [Refs. 5,18].

#### ***b. Advantages***

The advantages of incremental logging are two-fold. First, a system that runs with this method has the fastest processing time of database operations. Because the database is instantaneously updated with the logging operations, there is virtually no performance degradation of the system. And Secondly, the algorithm is straightforward and therefore easy to implement. This algorithm gambles that there will be very few roll-backs.

### *c. Disadvantages*

Although, incremental logging has the fastest processing time of the three algorithms, it is also the slowest of the three in roll-back and recovery. The primary reason for this is that this algorithm directly changes the database. In the event of a roll-back, the system must go out and change the database. This requires that all records updated by the transaction be deleted from the database and all records in the incremental log be retrieved and inserted into the database. After these two changes have been made, the system must then delete the record occurrences from the incremental log. This requires several disk accesses and is therefore very slow.

### **2. Differential Files**

Where incremental logging kept a file of records that were changed and instantaneously modified the database, a *differential file* is a log where all the updates to the database are deferred until the end of the transaction. The file also consists of an area along with an area processor for data manipulation that is separate from the area processors assigned to the database areas. When a transaction needs to update the database, a copy of the selected records is transferred into the differential file. All of the updates are modified in the differential file and at the end of the transaction all updated records are merged into the database. In addition, the records in the differential file are in actuality part of the database, subject to processing by all queries and therefore require additional bits for marking the records in

the differential file. These additional bits identify which is the most recent version of a record. For example, if the extra bit is 0, then the system knows that the most recent version of the record still resides in the main database and therefore the differential file is skipped and the main database is accessed immediately. To get the most current record, the system searches the differential file for every record retrieval except in the case where the extra bit is 0, then the original copy is accessed from the main database.

*a. Algorithm Description*

Figure 5, depicts the placement of the differential file if it were placed in MBDS. As each transaction enters the controller, the system logs the transaction into the differential file. When the transaction completes, the transaction is sent to the backends to merge into the database.

Specifically, as with incremental logging, when a transaction enters the system, a "start" record is written to the differential file with the transaction number attached. During the execution of the transaction, any write operations are entered into the file. As a minimum, a record consists of a transaction name, data item name, and new value of the data. When the transaction partially commits, a "commit" record is written into the file. After the transaction partially commits, each partially committed record is written to stable storage in case of a system failure. After the transaction is completely finished (no more requests), then the actual updating to the database can take place. Once the transaction has been entered into the

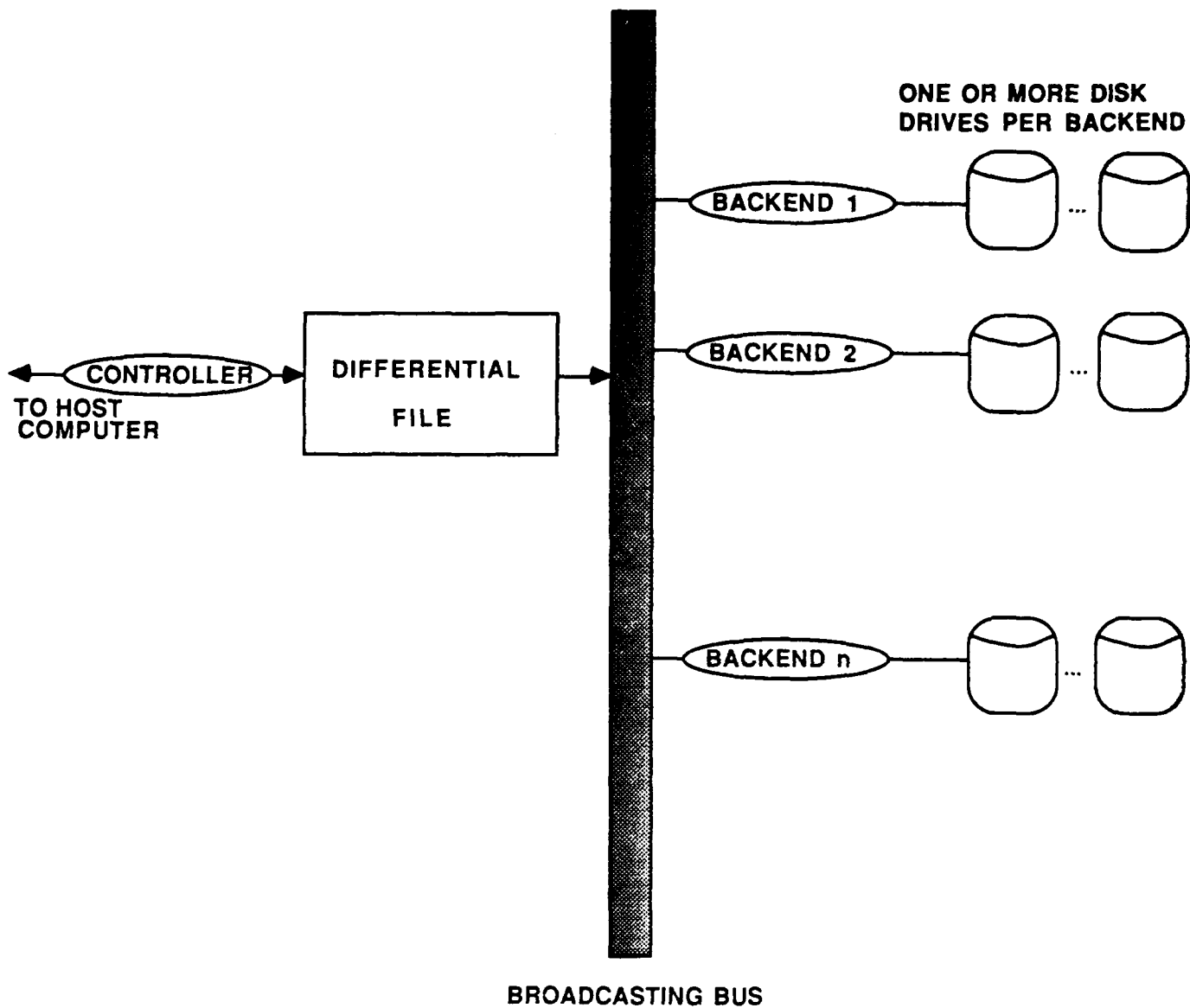


Figure 5. Differential File on MBDS Hardware Organization

actual database then it enters the committed state. After the transaction enters the committed state and a checkpoint has been reached, then the entire transaction is deleted from the differential file.

Where the incremental log had two primary functions, the *redo* and *undo*, the differential file has one function; the *redo*. Because the system is not updating the database directly, the only command that needs to be executed is one that repeats the submission of a transaction to the database from the differential file. The *redo* function sets the value of all data items updated by the transaction to the new values. In order to guarantee correct behavior even if a failure occurs during the recovery process, the *redo* function must be able to execute many times but the net effect must be equivalent to executing it only once.

#### ***b. Advantages***

The primary advantage of differential files is that it isolates the database from physical change by directing all new and modified records onto a separate and relatively small file of changes. Since the main database is never changed in the middle of a transaction, the system can quickly recover from a system failure or roll-back. In other words, transaction aborts are easily handled by simply discarding the record copy in the scratch area. Aghili and Severance [Ref. 4] note that a differential file architecture offers an approximately 77 percent improvement in recovery speed when compared to a update in-place algorithm (incremental log). For a precise

mathematical analysis of the differential file model see Aghili and Severance [Ref. 4].

### *c. Disadvantages*

Since differential files do not update the database directly, its major disadvantage is that it requires the additional overhead of reading differential file pages and the extra CPU overhead to process a query. For a system that seldom has a need for roll-back, the overhead of a differential file may not be justified.

## **3. Shadow Paging**

As stated earlier, Shadow Paging is an alternative to a log-based recovery technique. The database is partitioned into a number of fixed-length pages. The pages are not stored in physical order on the disk. Instead, the pages are indexed by the use of a page table. The page table has an entry for each database page stored on disk. The physical order of the page table depicts the logical order of the data on disk. During a transaction a page has a current page table and the shadow page table. All updates are made to the current page table with the shadow page table as the recovery source in case the transaction fails.

### *a. Algorithm Description*

When the transaction starts, the shadow page table and the current page table are identical. During the entire duration of the transaction, the shadow page table is not changed. The current page table, however, is

changed when a transaction performs a write operation. All of the input and output operations use the current page table to locate the database pages on the disk. During a write operation, if the page required is not in main memory, then the page is brought into main memory. If this is the first write performed on this page, then the system must find an unused page on disk (look at free page list), delete the unused page from the free page list, modify the current page table so that it now points to the new page, and then copy and modify the old page with the write. When the transaction commits, the current page table is written to disk (nonvolatile storage) and then becomes the shadow page table. In the event of a system crash or transaction abort, the old shadow page still resides on disk and therefore will restore the system to its last consistent state.

#### *b. Advantages*

Shadow paging offers some advantages over log-based systems. First, shadow paging eliminates the overhead of logging records. And secondly, according to Korth and Silberschatz [Ref. 18] recovery is significantly faster with shadow paging than with log-based systems.

#### *c. Disadvantages*

Although, shadow paging has some significant advantages, its disadvantages eliminate it from further consideration. First, data becomes extremely fragmented because the pages change location every time they are updated. This creates a significant change in the physical locality of the

pages. Since we want related pages close to one another on disk, we have violated the locality principle by requiring the system to travel a great distance (relatively speaking) to find the related data. Gray [Ref. 3] found that shadow paging was bad for direct (random) processing and for sequential processing for the reason stated above. The time savings in recovery have been negated by the reduction in system performance or a requirement for even higher-overhead schemes for physical storage management.

Figure 6 depicts the problem of violating the locality principle. Note that in the shadow page table, page number 4 has been modified by the current transaction and the current page table is now pointing at the 6th page on the disk as compared to the 4th page. Once the transaction commits, the current page table becomes the shadow page table and the physical relationship between the pages no longer exists. What exists now is a logical relationship which requires more disk access time. This speaks nothing of the fragmentation that is going on as pages become inaccessible which is the next problem that is addressed.

The second problem with shadow paging is that after each transaction commits, the old page is lost and becomes inaccessible (i.e., not free space and no useable information). This creates a severe garbage collection problem for the system. Garbage collection will add even more overhead and complexity to the system. Additionally, in order to use a



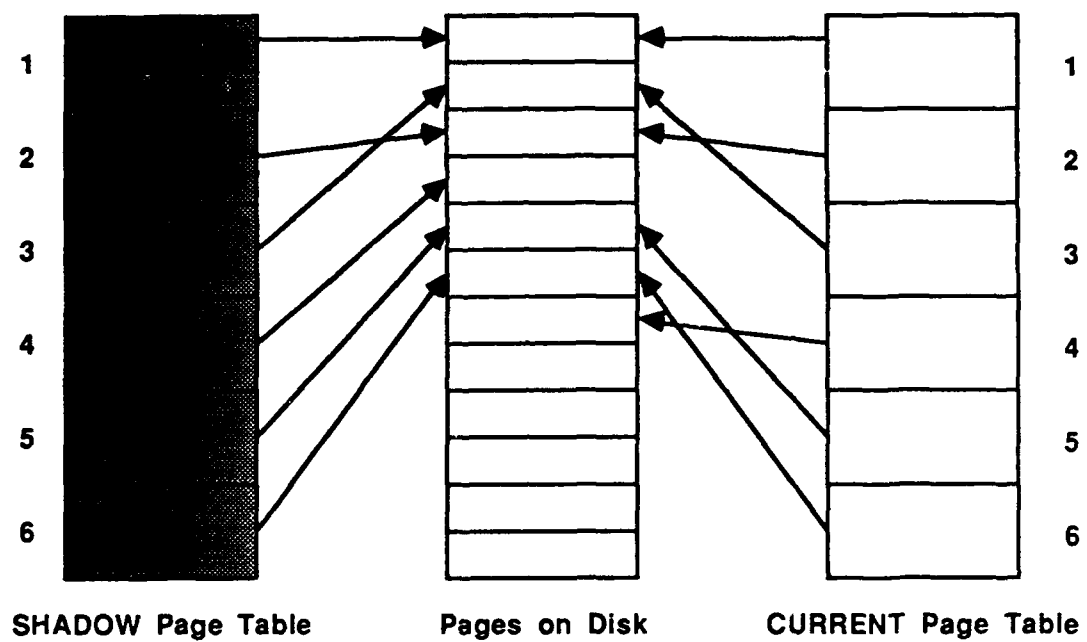


Figure 6. Shadow and Current Page Tables

shadow mechanism, the system requires a large amount (20 percent) of disk space to hold the shadow pages [Ref. 3].

And the last problem with shadow paging is that it is more difficult than logging algorithms to adapt to a concurrent operating environment (i.e., more than one transaction occurring a one time). In order for shadow paging to be used in a concurrent environment, the system would have to use a log. The use of a log would negate the advantage of having a shadow paging system in the first place. Additionally, the level of granularity is restricted to the size of a page which further hinders its performance in a concurrent environment. IBM's first relational system, System R, employed a recovery manager that used shadow paging. By using shadow paging, they also were forced into the use of an incremental log. Gray states [Ref. 3, p. 231] that they were unable to architect a transaction mechanism based solely on shadows which supported multiple users and in retrospect wished they had used a log-based recovery system with no shadow paging. He went on to say that the shadow page table was redundant and became extremely expensive for large files.

### C. SUMMARY AND CONCLUSIONS

In summary, the three general algorithms for implementing roll-back and recovery were *incremental logging*, *differential files*, and *shadow paging*. With *incremental logging* the updating of the database is done immediately with a log used to recover the system in case of roll-back. With *differential*

*files*, the database is not updated until the transaction has completely finished and a log of the update has been entered into the file. And finally with *shadow paging*, there are two copies of the page table. The page table that keeps track of the current updates is called the *current page table*. The page table that maintains the old version of the page table is called the *shadow page table*. After a transaction has been completed, the current page table becomes the shadow page table.

A review of the three general algorithms has given a good basis for deciding on the general algorithm to use in implementing the roll-back and recovery mechanism for the MBDS system. As discussed in the last section, the algorithm for shadow paging has been discarded for the primary reason that it is very difficult to implement in a concurrent transaction environment. The overhead required to implement shadow paging in a concurrent operating environment as MBDS would not be cost effective. With that decision made, we now turn our attention to the first two algorithms; incremental logging and differential files.

Before completely turning our attention to the two methods, we must first review our primary objectives in implementing a roll-back and recovery mechanism. As stated in Chapter 1, Section A.1, a very powerful extension of roll-back and recovery would be the ability to insert a transaction, allow the supervisor to see the results of that transaction, and then if he/she liked the results, have the transaction committed to the

database. If the results were unsatisfactory to the supervisor, then the supervisor could invoke what we call a *supervisor roll-back*. The ability of the supervisor to roll-back would require that a recovery system be prepared to roll-back numerous times. Therefore we need a recovery mechanism that maximizes roll-back and recovery speed and at the same time performs in a manner that does not degrade normal system performance.

In the first general algorithm, incremental logging, we found that the advantages of incremental logging are two-fold. First, a system that runs with this method has the fastest processing time of database operations. Because the database is instantaneously updated with the logging operations, there is virtually no performance degradation of the system. And Secondly, the algorithm is straightforward and therefore easy to implement. However, although incremental logging has the fastest processing time of the three algorithms, it has one major disadvantage, it is the slowest of the three in roll-back and recovery. Cardenas [Ref. 5] and Agrawal [Ref. 9] agree that in cases where there may be many roll-backs or other circumstances that incremental logging would not be the best selection. They go on to state that the recovery actions of rollback and restart are performed faster with differential files or shadow paging (with a log). For a statistical analysis and comparison of all three cases refer to Cardenas [Ref. 5] and Agrawal [Ref. 9].

On the basis of the empirical evidence listed above and the requirements of the system design, we have selected a modified version of the *differential file* approach as our roll-back and recovery mechanism for MBDS.

### **III. IMPLEMENTATION ISSUES**

#### **A. IMPLEMENTATION CONSIDERATIONS**

##### **1. Problem Specification**

In developing an algorithm to roll-back and recover MBDS, there are primarily four concerns that must be addressed. First, we want the user to be able to put in a test transaction and to see the results of those changes. Since the user could do this many times, we would like to roll-back efficiently by permanently changing the database only when the user wants to commit the transaction. Secondly, we want to develop in the algorithm the ability to commit or uncommit the transaction based on the user's desires. Thirdly, we must develop a data structure that retains the current state of the database but is also able to integrate the new test data into queries without changing the database. And fourthly, we have to slightly modify the concurrency control mechanism to maintain a consistent view of the database.

##### **2. Modifications to the Differential Log**

In a traditional differential file algorithm, the system logs every transaction and then later enters the transaction into the database. Before the transaction is entered into the database, the system treats the log as part of the database. In concurrent operations, other simultaneous transactions must be able to reference both the differential file and the physical database.

In the MBDS algorithm, the differential log (*Backend Transaction Log* (BTL)) only logs those transactions which start with BEGIN and end with END. The system responds to any RETRIEVE requests in the transaction by first querying the database for the information and then goes to the BTL to make any changes to the RETRIEVE request. The transaction is not sent from the BTL to the database until the system receives a COMMIT from the user. If the system receives an UNCOMMIT, then the transaction is flushed from the BTL and no change to the database is needed.

The reasons for this type of log are straightforward. First, once a test transaction enters the system, the user only receives the information from the system by placing RETRIEVE's in the transaction since all modifications (UPDATE, INSERT, DELETE) to the database are transparent to the user. Once all the information from the RETRIEVE's has been gathered into a buffer location from the database, all the modifications can be done in the buffer location based on the transaction in the BTL. Therefore all modifications are done in the buffer location and not to the database. Secondly, the concurrency control mechanism does not have to change except in the case of a TEST. In the case of a TEST, there will be the need for additional read LOCKs into the BTL. The reason for the additional locks would prevent a concurrent process from writing over or deleting a TEST transaction inadvertently. Thirdly, minimal changes are required to the

system to implement the algorithm. And finally, the system incurs very little additional overhead with the system upgrade.

### **3. The Placement and Description of the Log**

The two sections that follow describe the two possible locations that the BTL could be placed. In the centralized version, the BTL is placed at the controller level and centrally manages the test transactions. In the decentralized version, a copy of the BTL is placed in each one of the backends.

In either case, the BTL would abstractly look like the table in Figure 7. Each transaction would be assigned a transaction number. Since each transaction is made up of a number of requests, a request number is assigned to each atomic ABDL command. The ABDL command is then put into the table. The next field is used to state whether the request is a test or not. Although this field does not seem necessary since the request would not be in the table if it were not a test, it is a very important part of the data structure because it tells the system that this is a test request. There will more on this in Chapter IV. And finally, the last two fields give the status of the request (i.e., committed or uncommitted), and the user number.

### **B. THE CENTRALIZED APPROACH**

In this section, we present the centralized placement of the BTL in the system controller. The first section describes the user interface to the system. The second section describes the system algorithm to implement the



TRAN #	REQ #	ABDL COMMAND	TEST	COMM	USER #
T <sub>0</sub>	R1	INSERT(<File,Census>,<Population,58000>,<City, Sly>),	YES		1
	R2	RETRIEVE (CITY $\neq$ BOSTON) (CITY),	YES		"
	R3	UPDATE(File=Census and City=Cumberland)(.... )	YES		"
T <sub>1</sub> ...	R1				
	R2				
	R3				
	R4				
...					
...					
T <sub>N</sub>	R1				
	R2				

Figure 7. Backend Transaction Log

centralized approach. And then the last two sections describe the advantages and disadvantages of this algorithm.

### **1. The User Interface**

The user interface is depicted in Figure 8. In Figure 8 the user would run his MBDS software and see a set of choices:

- 1. COMMIT**
- 2. TEST**
- 3. QUIT**
- 4. DATA DICTIONARY**

The COMMIT selection would tell the system that roll-back will not be required and to function in a normal manner (make changes to the database as necessary). The QUIT selection will exit the user to the operating system. The DATA DICTIONARY selection would give the user index information into his database. If the user selects TEST, he will be prompted to enter BEGIN and then insert the transaction. At the end of the transaction, the system again prompts the user to insert the END into the transaction. The system then inserts the transaction into the BTL. After the transaction has been inserted into the BTL, the system searches for any RETRIEVE requests in the transaction and sends those requests to the backends. The backends receive the requests and check to see if the requests are for a TEST, if a request is for a TEST, the backend gathers the information from the database and then requests a read LOCK on the BTL. Once the backend gets access

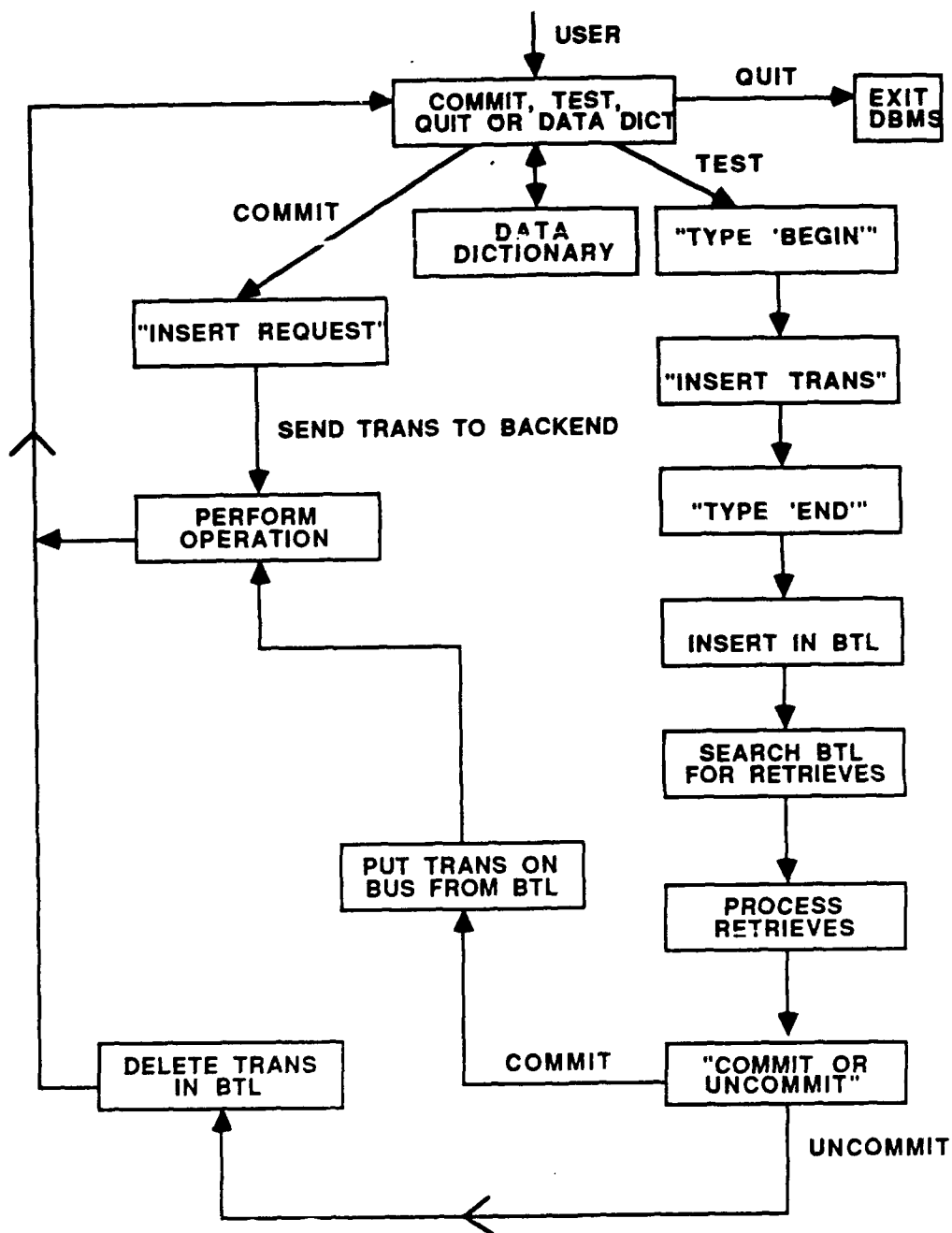


Figure 8. Centralized BTL User Interface

to the BTL, it modifies the information gathered from the database and sends the results to the controller. The user is then given the information requested. The user can then COMMIT or UNCOMMIT the transaction. If the user commits, the transaction (minus the RETRIEVE's) is to be processed by the system in the normal manner. If the user uncommits, the transaction is simply flushed from the BTL. In either case, the user is returned to the initial user interface for further processing. In the following section, the general algorithm is discussed in greater detail.

## 2. The General Algorithm

Figure 9 depicts the relationship between the hardware and software in MBDS. The top block is the controller to MBDS. The bottom block is one backend to the MBDS system, understanding that there could be many backends to the controller. Additionally, we have placed the BTL in the controller in this centralized model.

Initially the transaction is received by the controller from the user (Figure 9,(1)). The *Request Preparation Section* of the controller formats the user requests into a table. The user then is given a choice to either COMMIT, TEST, DATA DICTIONARY, or QUIT. If the user selects COMMIT, then the transaction is sent directly to the *Directory Management Section* of the backends (Figure 9,(2)).

If the user selects TEST, then all the operations are to be logged into the BTL. The BTL manager will send out the RETRIEVE's in sequential

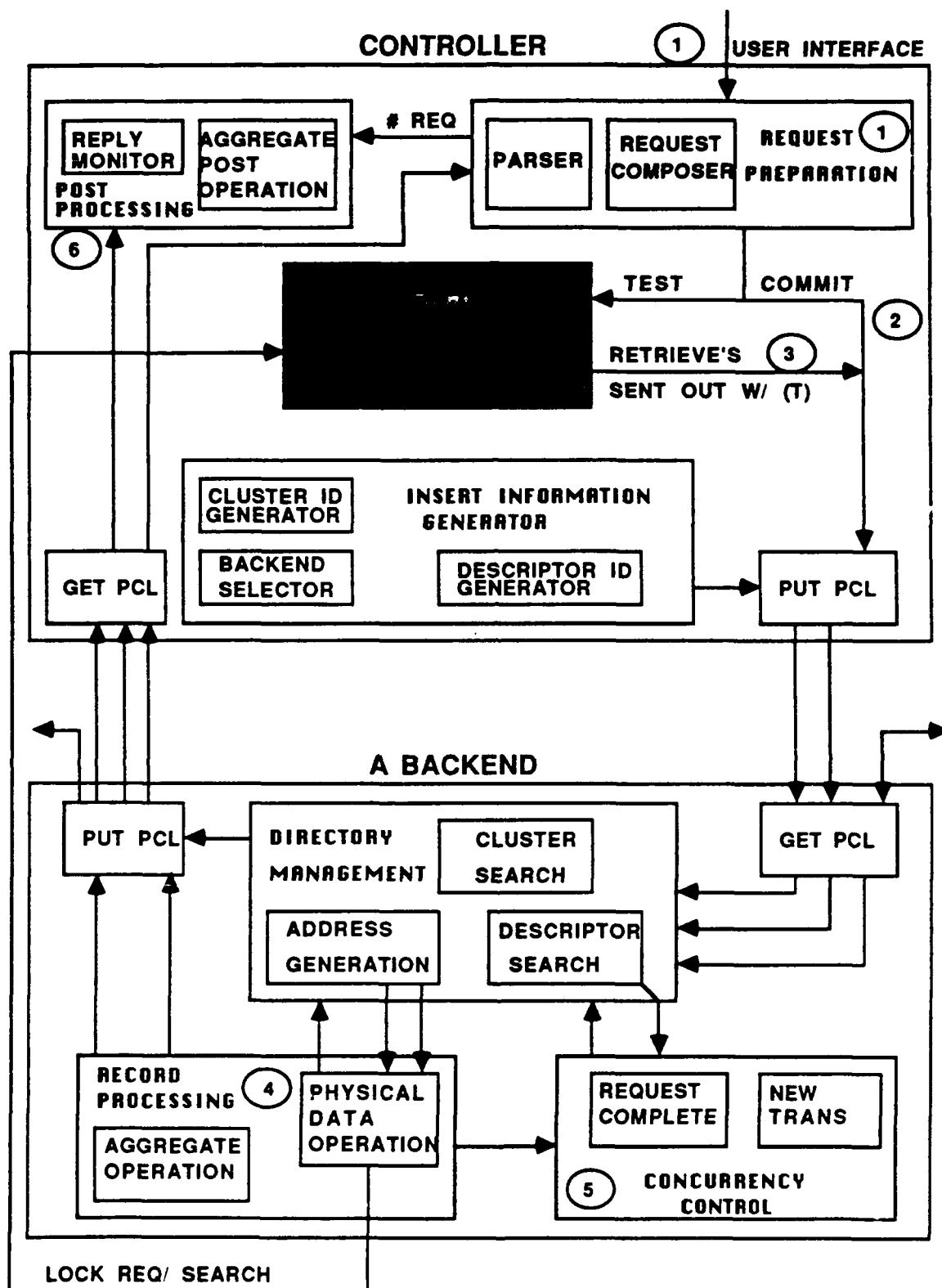


Figure 9. Centralized Backend Transaction Log (BTL)

order (Figure 9,(3)). The test RETRIEVE's will be specially designated in their fields as TEST queries. The backends will be required to not only check their databases for the information but to look into the BTL for any updates/insertions/deletions to the transaction requests. Each backend will require a read LOCK in order to access the BTL. These LOCKs will be managed by the BTL manager and requested by the *Record Processing Section* of the backends (Figure 9,(4)). The BTL LOCKs' will be managed by the BTL manager and the LOCKs to the database are managed by the *Concurrency Control Section* (Figure 9,(5)) of the backends. For more information on the MBDS concurrency control mechanism refer to [REFS. 14,15]. The read LOCKs in the BTL will only be given to those requests whose transaction and request number are equal to the those in the BTL. The updated user request of the test RETRIEVE will be sent directly to *Post Processing* (Figure 9,(6)) for dissemination to the user.

The user is then given the choice to COMMIT or UNCOMMIT the test. If the user commits the transaction, then all requests of the transaction (except the RETRIEVE's) are sent to the backends from the BTL for processing. The entries in the BTL of the transaction are deleted from the BTL. If the user uncommits, then all requests for a given transaction in the BTL are deleted.

If the user selects DATA DICTIONARY, the user is given information about his indices into the database. And finally, if the user chooses QUIT, he will be exited from DBMS to the operating system.

### **3. Advantages**

There are primarily three advantages to using the centralized BTL. The first reason is that a copy of the BTL is located centrally so there is very little overhead to the total system. The second reason is that assuming that the majority of the transactions will commit directly, there is no change to the current system. And finally, the locking mechanism will only have to change for "tests" which will occur only a small percentage of the time in relation to the total users on the system.

### **4. Disadvantages**

The one major disadvantage of this system is that this would significantly increase the traffic over the network. The additional traffic over the network would cause processing delays and degradation of system performance. In addition, the major design goal of maximizing the work of the backends would be violated by requiring the controller the additional performance overhead of maintaining the BTL.

## **C. THE DECENTRALIZED APPROACH**

In this section, we present the decentralized placement of the BTL in each one of the backends. The first section describes the user interface to the system. The second section describes the system algorithm to implement

the decentralized approach. And then the last two sections describe the advantages and disadvantages of this algorithm.

### **1. The User Interface**

The user interface to the decentralized BTL is very similar to the centralized BTL. The decentralized BTL user interface is depicted in Figure 10. In Figure 10, the user would run his MBDS software and see the following set of choices:

- 1. COMMIT**
- 2. TEST**
- 3. QUIT**
- 4. DATA DICTIONARY**

As in the centralized approach, the COMMIT selection would tell the system that roll-back will not be required and to function in a normal manner (make changes to the database as necessary). The QUIT selection will exit the user to the operating system. The DATA DICTIONARY selection would give the user index information into his database. If the user selects TEST, he will be prompted to enter BEGIN and then insert the transaction. At the end of the transaction, the system again prompts the user to insert the END into the transaction. The system then inserts the transaction into the BTL. The primary difference here between the centralized approach and the decentralized approach is that instead of making 1 copy of the transaction and storing it in the controller, the decentralized model broadcasts the



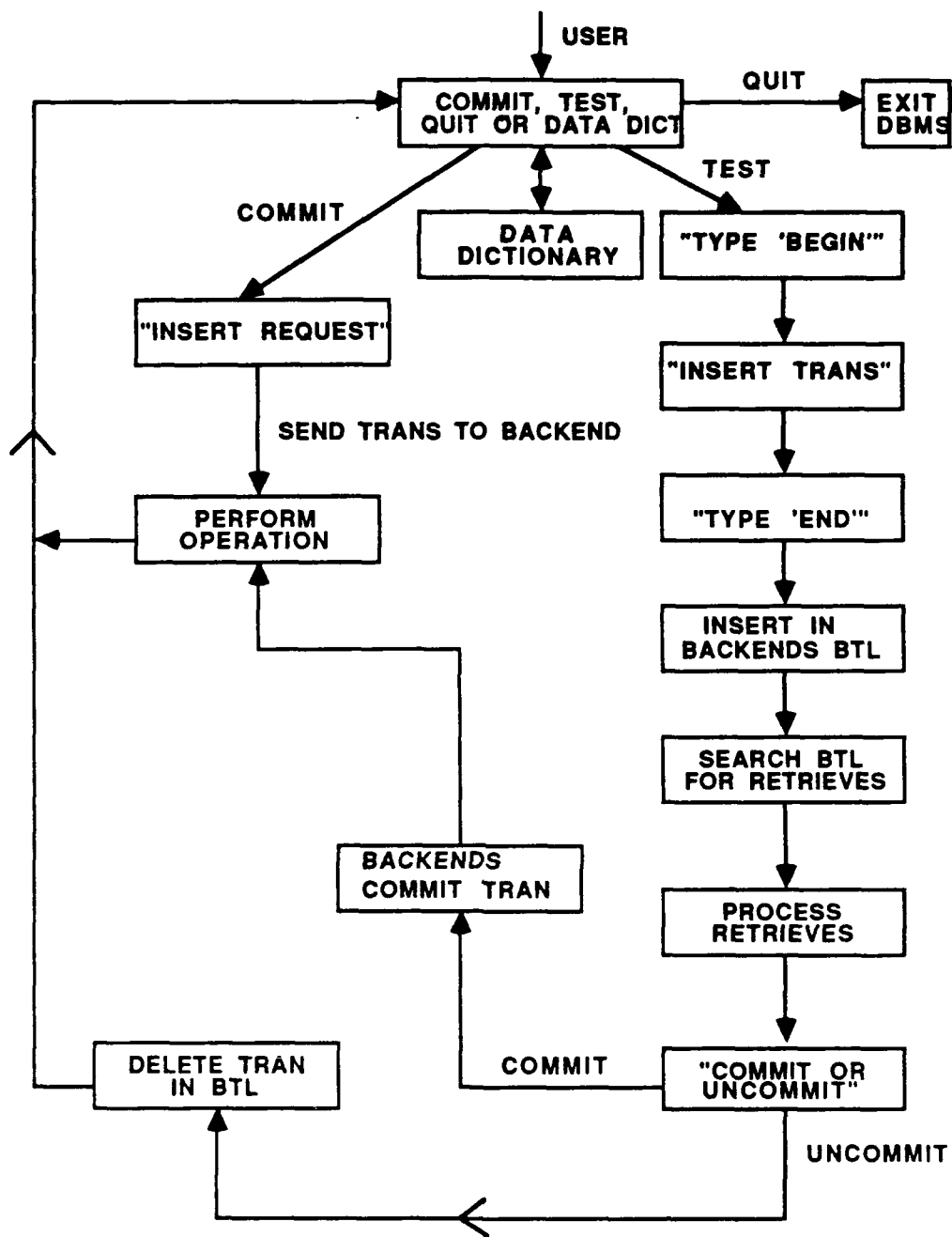


Figure 10. Decentralized BTL User Interface

transaction to all the backends to be entered into the BTL. When the transaction is received, the backend checks to see if it is a TEST, if it is a TEST, the backend inserts the transaction into the BTL. After the transaction has been inserted into the BTL, the local BTL searches for any RETRIEVE requests in the transaction and immediately accesses the database to gather information. The backend then requests a local read LOCK on the BTL and modifies the requested information based on the information in the BTL. This information is passed on to the controller who compiles the information from all the backends and sends the requested information to the user. The user can then COMMIT or UNCOMMIT the transaction. If the user commits, the transaction (minus the RETRIEVE's) is sent to be merged into the database in the normal manner. If the user uncommits, the transaction is simply flushed from the BTL. In either case, the user is returned to the initial user interface. In the next section, the general algorithm is discussed in greater detail.

## **2. The General Algorithm**

Figure 11 depicts the relationship between the hardware and software in MBDS. As describe earlier, the top block is the controller to MBDS. The bottom block is one backend to the MBDS system, understanding that there could be many backends to the controller. The significant change here is that now the BTL is located in each backend.

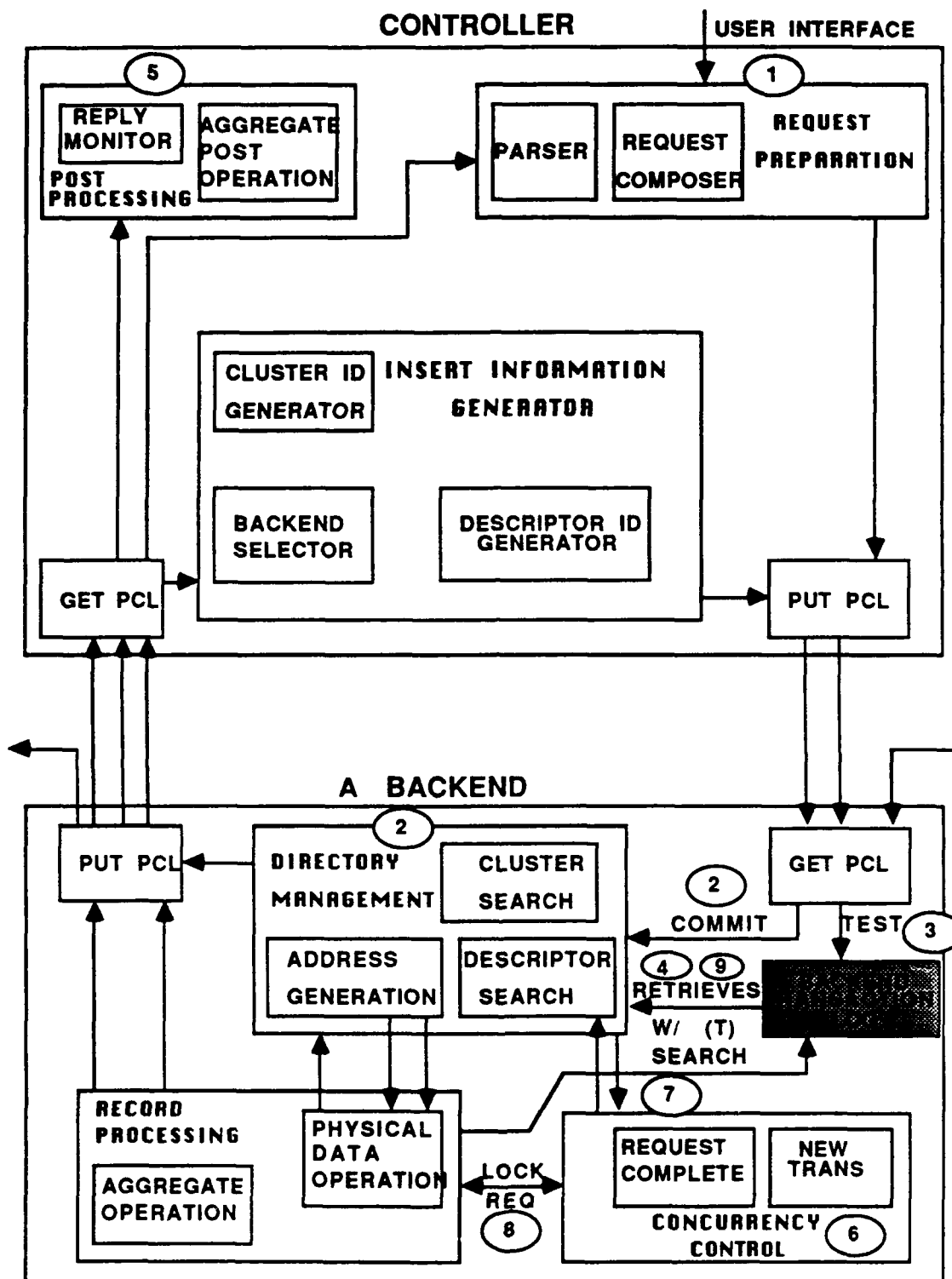


Figure 11. Decentralized Backend Transaction Log (BTL)

Initially, the transaction is received by the controller from the user (Figure 11,(1)). The *Request Preparation Section* of the controller formats the user requests into a table. The user then is given a choice to either to COMMIT, TEST, DATA DICTIONARY, or QUIT. If the user executes a COMMIT or a TEST, the transaction is sent directly to the backend. If the user selects COMMIT, then the transaction is sent directly to the *Directory Management* section of the backends (Figure 11,(2)).

If the user selects TEST, then the backend immediately logs the transaction into the BTL (Figure 11,(3)). The local BTL manager will send out the RETRIEVE's in sequential order (Figure 11,(4)). The test RETRIEVE's will be specially designated in their record fields as TEST queries. The backends will be required to not only check their databases for the information but to look into their local BTL for any updates/insertions/deletions to the transaction requests. The backend information, or a negative response (in the case that the backend has no information in the database relating to the query), is then sent directly to *Post Processing* (Figure 11,(5)) for compilation by the controller and disseminated to the user.

Each backend will require a read LOCK in order to access the BTL and additional read LOCKs to access the database. The BTL LOCKs will be managed by the local BTL manager and the LOCKs to the database are managed by the *Concurrency Control Section* (Figure 11,(6)). The LOCKs

to the database are requested by the *Directory Management Section* of the backends (Figure 11,(7)) and the LOCKs to the BTL are requested by the Record Processing Section (Figure 11,(8)). The read LOCKs in the BTL will only be given to those requests whose transaction and request number are equal to the those in the BTL. Once a transaction has been processed by an operation, the LOCKs are then given back to the BTL manager and *Concurrency Control Section*.

The user is then given the choice to COMMIT or UNCOMMIT the TEST. If the user commits the transaction, then all requests of the transaction (except the RETRIEVE's) are sent to the *Directory Management Section* (Figure 11,(9)) for processing. The entries in the BTL of the transaction are deleted from the BTL. If the user uncommits, then all requests for a given transaction in the BTL are deleted.

If the user chooses DATA DICTIONARY, the user is given information about his indices into the database. And finally, if the user chooses QUIT, he will be exited from DBMS to the operating system.

### **3. Advantages**

This version of the BTL has some significant advantages. First, since each backend has its own copy of the transaction log, it is much quicker to get into the BTL (no requirement to network to the controller in order to request locks as in the centralized BTL version). This greatly reduces the traffic on the network increasing system performance. Third, assuming that

the majority of the transactions will commit directly, there is no change to the current system. Fourth, the locking mechanism will only have to change for tests which will occur only a small percentage of the time. And finally, one of the major design goals for MBDS was to minimize the work done by the controller and maximize the work done by the backends [Ref. 15, p. 1]. The use of the decentralized BTL maintains the spirit of the MBDS major design goal.

#### **4. Disadvantages**

The primary disadvantage of this model is that every backend will have a copy of the BTL. This redundancy will create extra storage overhead for each backend. If the data structure for the BTL is dynamic, the overall cost to the system would be negligible.

### **D. SUMMARY AND CONCLUSIONS**

In this chapter, we discussed the four major implementation issues that concerned the inclusion of roll-back and recovery in MBDS. The four issues were:

- 1) Give the user the ability to put in a test transaction and to see the results of those changes.
- 2) Develop in the algorithm the ability to COMMIT or UNCOMMIT the transaction based on the user's desires.
- 3) Develop a data structure that retains the current state of the database but is also able to integrate the new test data into queries without changing the database.
- 4) Modify the concurrency control mechanism.

Additionally, the traditional differential file model had to be modified to fit the four issues listed above and MBDS. In the MBDS algorithm, the differential log (*Backend Transaction Log* (BTL)) only logs TEST transactions. The system responds to any RETRIEVE requests in the transaction by first querying the database for the information and then goes to the BTL to make any changes to the results of the RETRIEVE request. The transaction is not sent from the BTL to the database until the system receives a COMMIT from the user. If the system receives an UNCOMMIT, then the transaction is flushed from the BTL and no change to the database is needed.

There are two places to put the BTL. The first option, the centralized BTL, called for putting the BTL in the controller. The centralized BTL had primarily three advantages; little system storage overhead, small change to the current system, and the concurrency control mechanism would only have to change for tests. But, the Centralized BTL had two major disadvantages. First, this method would significantly increase the traffic over the network due to the need to receive read LOCKs from the BTL. Second, the major design goal of maximizing the work of the backends would be violated by requiring the controller the additional performance overhead of maintaining the BTL.

The second option, the decentralized BTL, called for putting the BTL in each one of the backends. This version of the BTL has some significant

advantages; quicker access to the BTL, less network congestion than in the centralized BTL, little change to the current system, locking mechanism will only have to change for tests, and finally, maximizes the work done by the backends. The primary disadvantage of the decentralized approach is the replication of the BTL in each one of the backends. This can be overlooked to some extent if the data structure used for the BTL is dynamic. Once the transaction is committed or uncommitted, it is flushed from the BTL and the system recovers the storage overhead.

On the account of the significant advantages of the decentralized BTL over the centralized BTL, MBDS will use the decentralized BTL log as its recovery storage structure.



## IV. THE TRANSACTION LOG

In this chapter, the data structure and algorithm is defined that will be used to implement roll-back and recovery in MBDS. In the first section, the data structure is proposed for the BTL. In the second section, the "big picture" of the algorithm is described from a decomposition point of view. After looking at the "big picture," the third section describes the algorithm in detail. And finally, in the last section I summarize the Chapter.

### 1. THE DATA STRUCTURE

An abstract view of the data structure that will contain the BTL is depicted in Figure 12. As shown in the figure, the BTL is basically an array of pointers that point to a linked list of records. For each index into the array, there is a unique and distinct transaction number associated with it. As discussed in Chapter I, every transaction is composed of a number of requests. Each request is put into a record and then inserted into the linked list of records. For the remainder of this discussion, requests and records are synonymous. When the logging of the transaction is finished, the user has a pointer from a array pointing to a linked list of records/requests.

When the transaction is sent to the backends, the BTL manager looks into the array of pointers, `POINTERARRAY`, and searches the array for the first `NIL` value. The index into `POINTERARRAY` is the transaction number and is placed into the first field of the record, `TRANSNUM`. The second field is

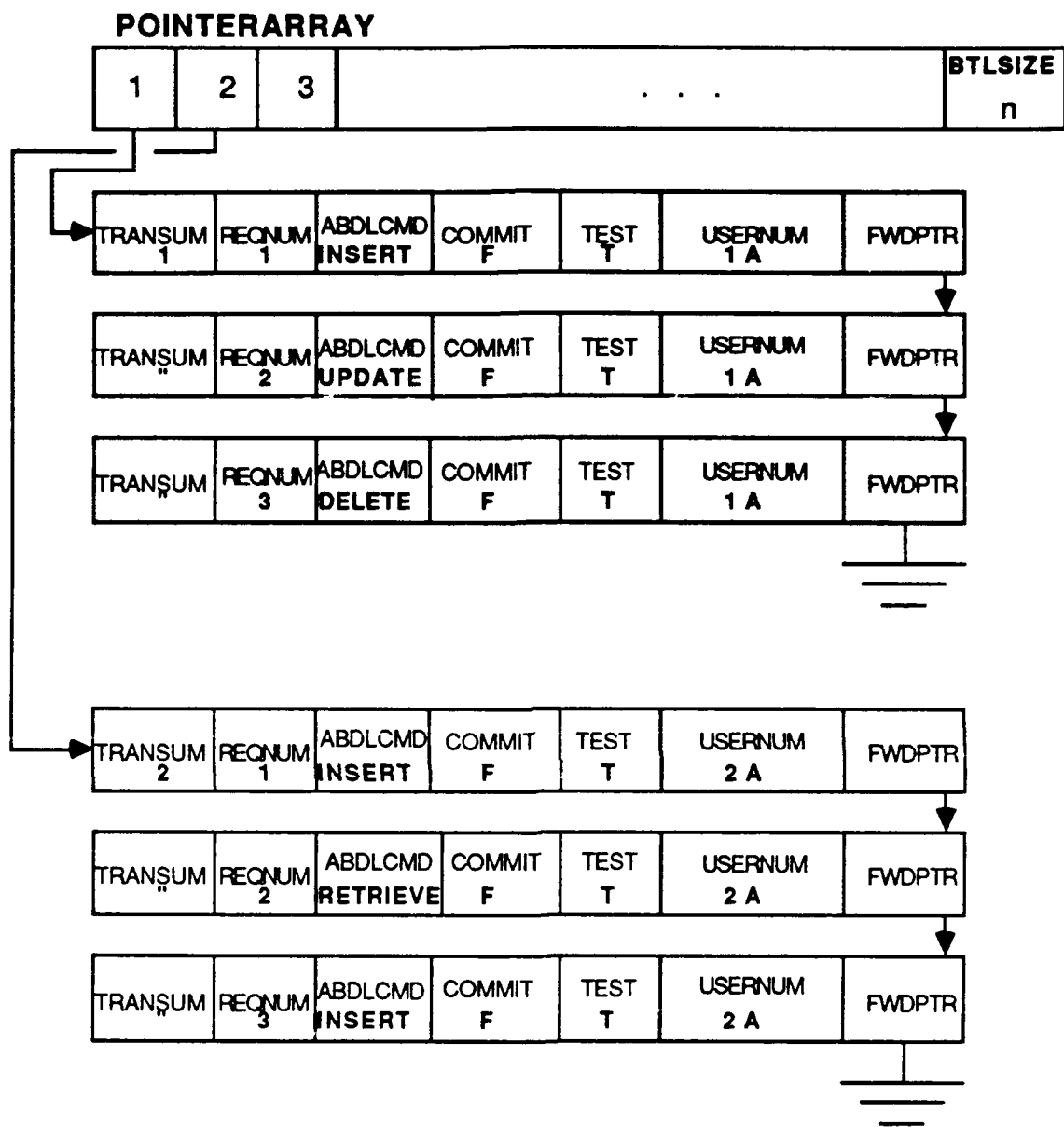


Figure 12. Backend Transaction Log Data Structure

the request number, REQNUM, which is the sequential order of the requests as they enter the BTL. The third field is the ABDL command, ABDLCMD, along with the arguments that follow the command. The fourth field is a boolean field, COMMIT, that sets the request to TRUE if the request is to be committed and to FALSE if the request is in an uncommitted state. When the transaction is first entered into the log, this field is set to FALSE. The fifth field is another boolean field, TEST, that tells the system that this is a TEST request. The sixth field, USERNUM, is the user's system identification. And finally, the last field, FWDPTR, is the pointer to the next request in the transaction's linked list or to NIL if it is the last request in the transaction.

Since this data structure is dynamic, there is very little overhead incurred on the backend. The only additional overhead to the backends in non-testing operations is the array of pointers which requires negligible storage space. During testing operations, the size of the BTL will be directly dependent on the number of users that are sending in test transactions.

## **2. A DECOMPOSITION VIEW OF THE ALGORITHM**

In Figure 13, the decomposition view of the algorithm is depicted. This algorithm is composed of 14 steps/procedures which are executed sequentially for each transaction once the system roll-back is invoked. When the user enters the system, he is presented with four choices D for data dictionary information, T for test (roll-back), C for commit the

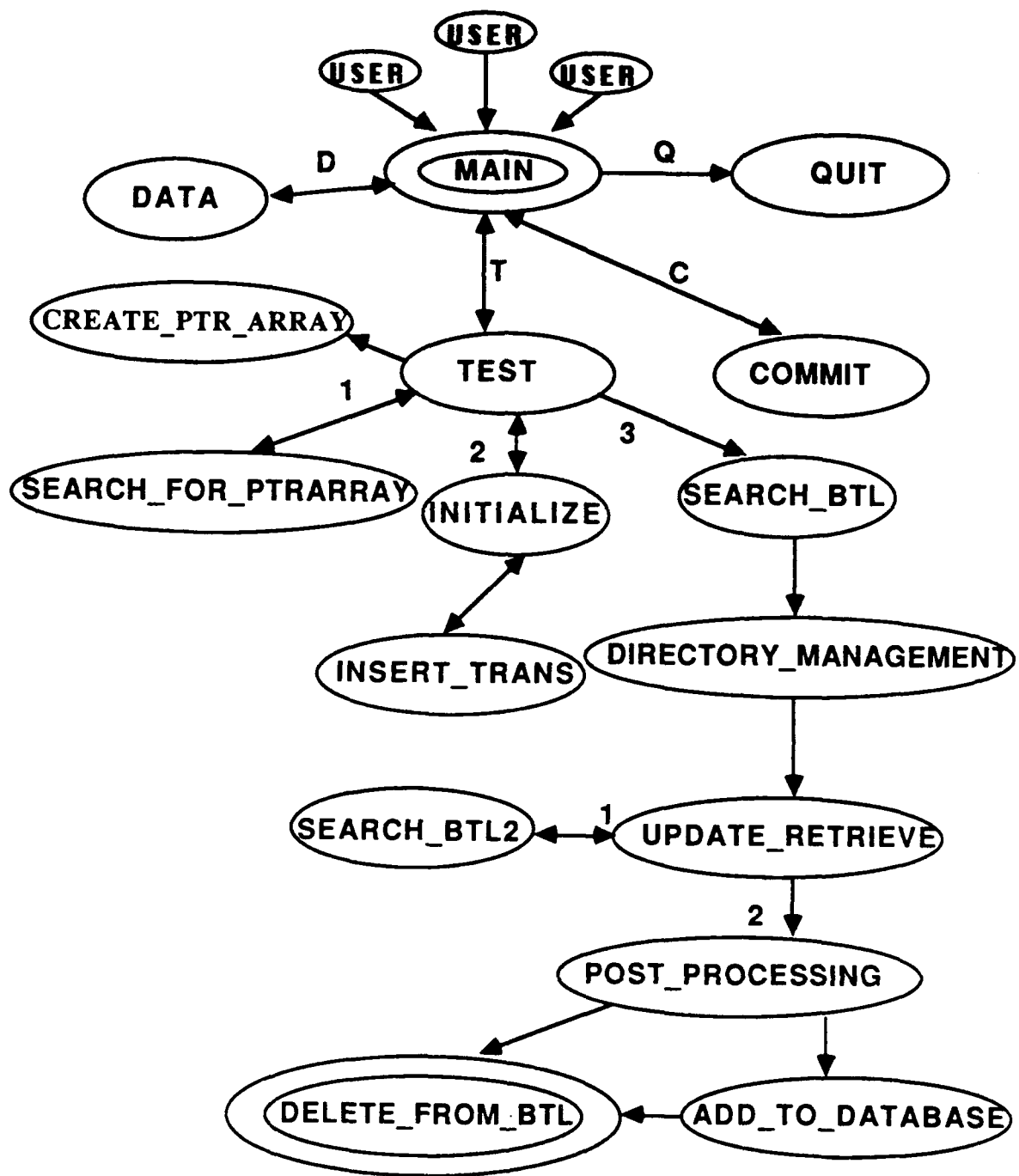


Figure 13. A Decomposition Diagram

transaction, or Q for quit and return to the operating system. In cases D, C, and Q, the system performs its operations in the usual manner [Refs. 14,15]. In the case where the user selects T, the roll-back and recovery mechanism is started.

Once the first user selects the test mode, the first step calls for an array of pointers to be created in each one of the backends (CREATE\_PTR\_ARRAY). This is the initial creation of the BTL. In the next step, TEST calls SEARCH\_FOR\_PTRARRAY in order to find the first pointer in the array that has a NIL pointer. This is the place where the transaction is to be inserted. In the third and fourth steps, TEST calls INITIALIZE which calls INSERT\_TRANS in order to create the record structure which inserts the request into the record, attaches the record to the pointer of the array, and then looks for the next request to insert into the data structure or BTL. This loop goes on until the entire transaction is entered into the BTL.

In the next step, the algorithm searches, SEARCH\_BTL, the BTL for any retrieves and sends those retrieves to DIRECTORY\_MANAGEMENT. DIRECTORY\_MANAGEMENT searches the database for the information and sends the information to UPDATE\_RETRIEVE if the request was a test. UPDATE\_RETRIEVE searches the BTL a second time to modify the information and then passes the information onto POST\_PROCESSING for dissemination to the user. If the user elects to COMMIT the changes then

POST\_PROCESSING sends a COMMIT message to ADD\_TO\_DATABASE. ADD\_TO\_DATABASE simply searches through the BTL for updates, inserts, or deletes and changes the TEST field from TRUE to FALSE and passes them to DIRECTORY\_MANAGEMENT to be added to the database. If the user selects to UNCOMMIT or after a transaction has been added to the database, the transaction is deleted from the database by DELETE\_FROM\_BTL. In the following section, the algorithm goes into greater detail.

### 3. THE ALGORITHM

In the Appendix, this algorithm is written in pseudo PASCAL for further reference. As discussed in section 1, the data structure of the BTL is an array of pointers where each pointer points to a linked list of records. The index into the array is the transaction and the linked list of records are the requests that compose the transaction.

In section 2, I described the initial user interface to the algorithm with the selection to COMMIT, TEST, DATA\_DICTIONARY, or QUIT. The present MBDS system remains the same for every user entry except for TEST. When the TEST option has been selected, the algorithm searches the array of pointers for a NIL pointer. SEARCH\_FOR\_PTRARRAY is the pseudo code algorithm for searching the array for the NIL pointer (see Appendix). Figure 14 displays an abstract view of the BTL during SEARCH\_FOR\_PTRARRAY. In this example, the system currently has

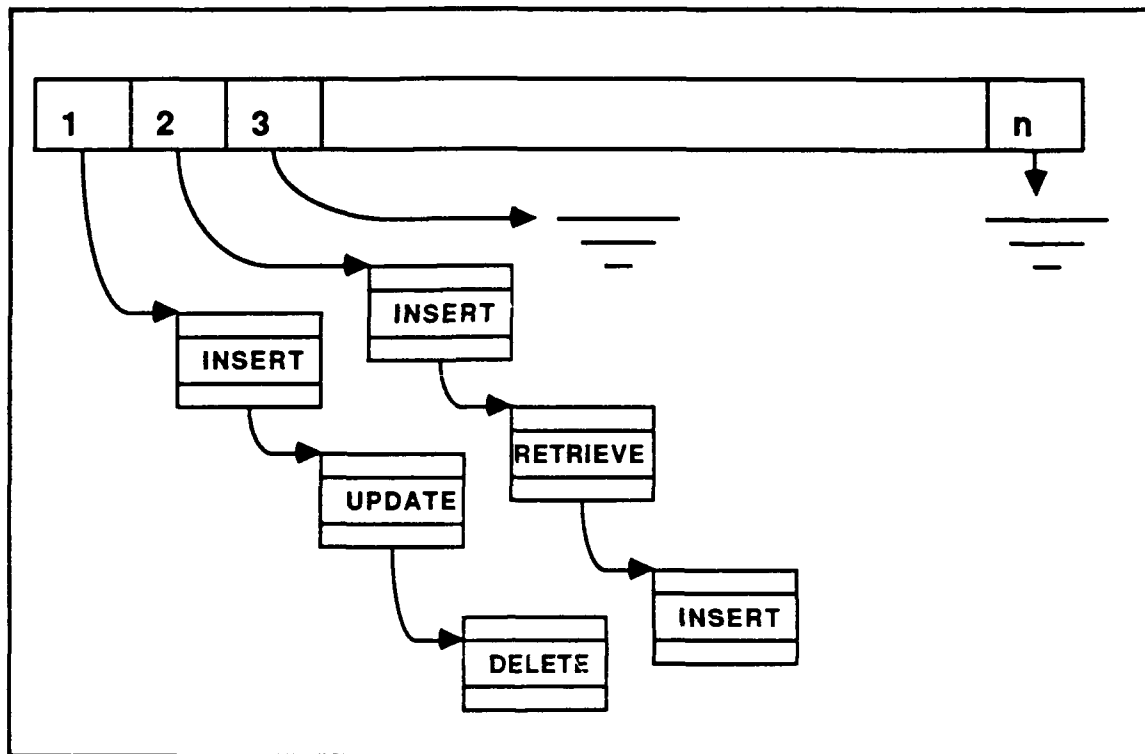


Figure 14. SEARCH\_FOR\_PTRARRAY

two test transactions active at indices 1 and 2. This procedure simply goes down the array until it finds that index 3 has a NIL pointer. This is where the next transaction will go. As indices 1 and 2 are completed, the transactions will be flushed from the BTL and their pointer's will return to NIL.

Having found a location in the BTL to insert the transaction, SEARCH\_FOR\_PTRARRAY sends the index of the array back to TEST. TEST sends the index of the array to INITIALIZE. INITIALIZE creates a record structure of the first request and then sends the request to INSERT\_TRANS where the request is attached to the pointer array. This loop is continued until all of the transaction has been inserted into the BTL. Figure 15 displays the loop as the transaction is inserted into the BTL. In the case of index 3, the first request inserted is an UPDATE. After the UPDATE, the algorithm gets the next request (RETRIEVE) and inserts it into the linked list. The initialization phase is completed after the last two UPDATES are inserted.

Figure 16 depicts the next stage of the algorithm. In Figure 16, the top of the figure abstractly portrays the software in the backend. At the bottom of the figure is the BTL data structure as it changes during the procedure SEARCH\_BTL. In order for the system to respond to TEST information from the user, it must be given the RETRIEVES from the BTL. In this phase of the algorithm, a temporary pointer goes through the transaction, in this



## INITIALIZE

RECORD

3	TRANSNUM
1	REQNUM
UPDATE	ABDLCMD
FALSE	COMMIT
TRUE	TEST
1 A	USERNUM
...	FWDPTR

THE BACK-END TRANSACTION LOG (BTL)

INSERT TRANS

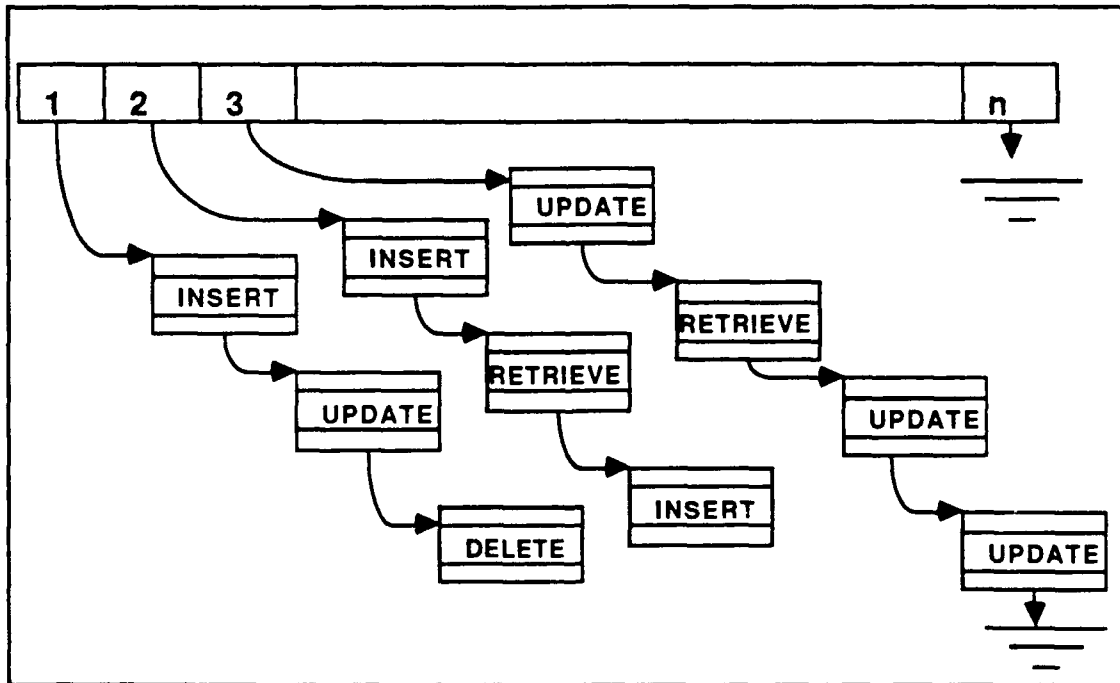


Figure 15. The BTL Initialization



case transaction 3, and finds the retrieve request and sends it over to Directory\_Management. The temporary pointer continues through the transaction until all retrieves have been sent to Directory\_Management.

DIRECTORY\_MANAGEMENT first checks to see if the RETRIEVE is a TEST RETRIEVE. If the RETRIEVE is not a TEST then DIRECTORY\_MANAGEMENT processes the transaction in the normal manner [Ref. 14,15]. But if the RETRIEVE is a TEST then DIRECTORY\_MANAGEMENT takes the following actions. First, DIRECTORY\_MANAGEMENT sends a request for type-C (dynamic) attributes needed by the RETRIEVE command to the Concurrency Control Section (CC). Once the attributes are locked, CC signals DIRECTORY\_MANAGEMENT. DIRECTORY\_MANAGEMENT next performs a descriptor search and signals CC to release the locks on the attributes. Following the descriptor search, DIRECTORY\_MANAGEMENT sends the descriptor-ID groups to CC. When the descriptor-ID groups are locked and the cluster search is allowed, CC signals DIRECTORY\_MANAGEMENT. DIRECTORY\_MANAGEMENT then performs a cluster search and signals CC to release the locks on the descriptor-ID groups. Following the cluster search, DIRECTORY\_MANAGEMENT sends the cluster-IDs for retrieval to CC. Once the cluster-IDs are locked and the request can proceed with address generation and the rest of the request execution, then CC signals

DIRECTORY\_MANAGEMENT. DIRECTORY\_MANAGEMENT then performs the address generation and sends the RETRIEVE\_REQUEST and the addresses to the Record Processing Section (RP). Once the RETRIEVE\_REQUEST has been generated, the Record Processing Section (RP) first checks to ensure that the request is a TEST, if the request is a TEST then RP requests a read LOCK on the BTL. When the LOCK is granted, Directory\_Management calls update\_retrieve.

As depicted in the bottom of Figure 17, UPDATE\_RETRIEVE creates a second temporary pointer and looks through the transaction in the BTL for any inserts, updates, or deletes. When inserts, updates, or deletes are found, the retrieve\_request is modified. The modified retrieve\_request is then sent to Post\_Processing Section of the controller where the transaction is aggregated from all the backends and displayed to the user.

The user is then given the option to COMMIT or UNCOMMIT the transaction. If the user selects COMMIT then ADD\_TO\_DATABASE is called. ADD\_TO\_DATABASE creates a third pointer into the transaction, Figure 18, that sends updates, inserts, and deletes to Directory\_Management. Before the requests are sent to DIRECTORY\_MANAGEMENT, the TEST field in the record is changed to false so that DIRECTORY\_MANAGEMENT treats the request as a normal request and enters the request into database. Once this function is performed, then RP informs CC that the request is done and the locks on the cluster ids can be

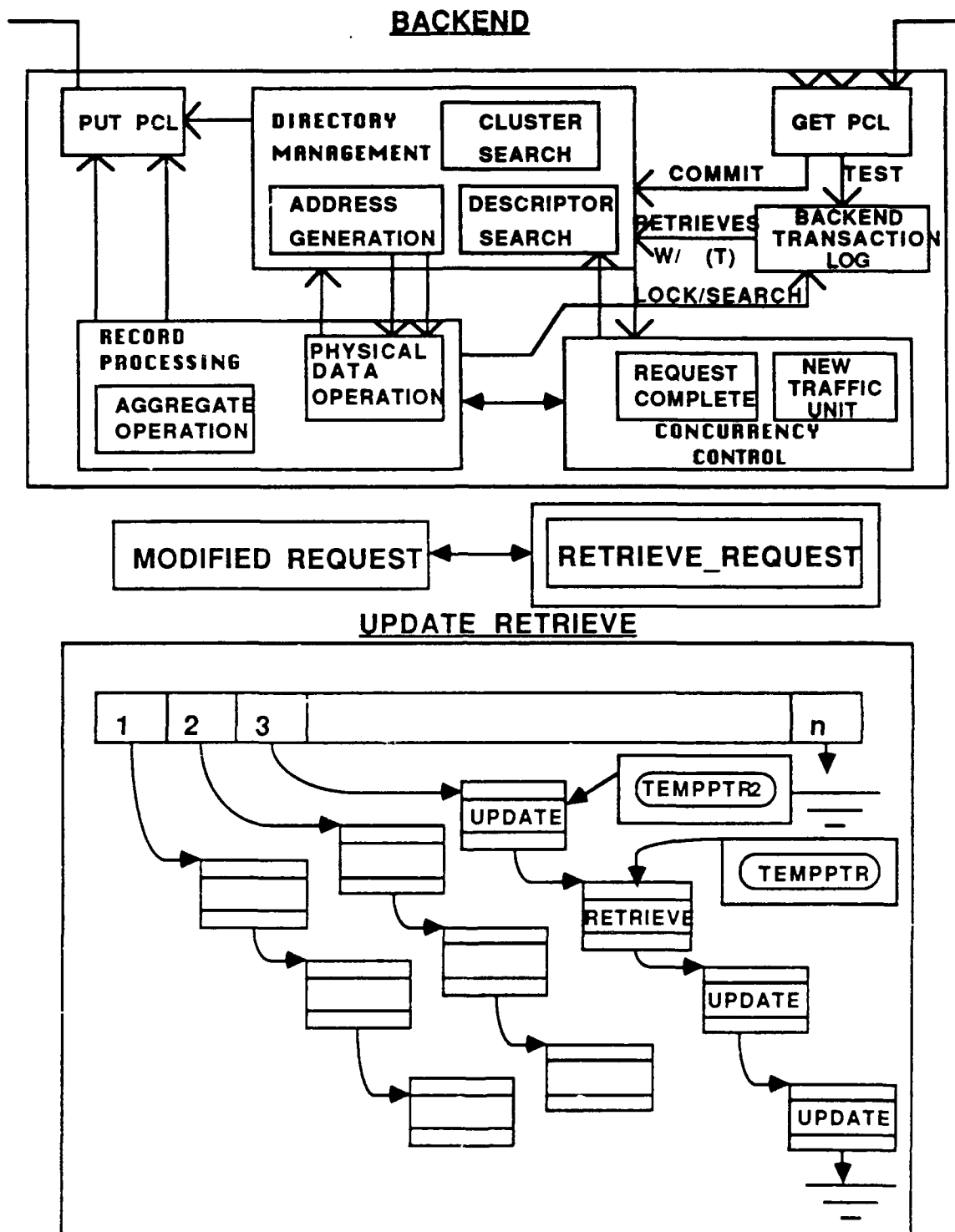


Figure 17. Update the RETRIEVE\_REQUEST

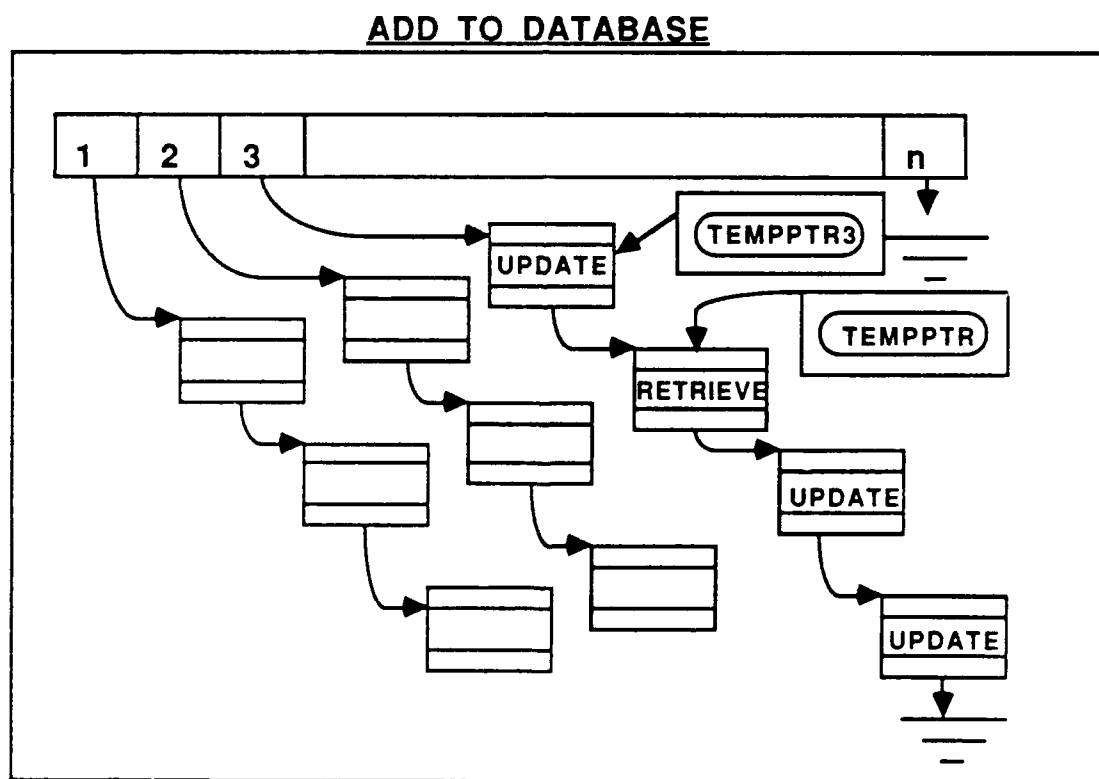
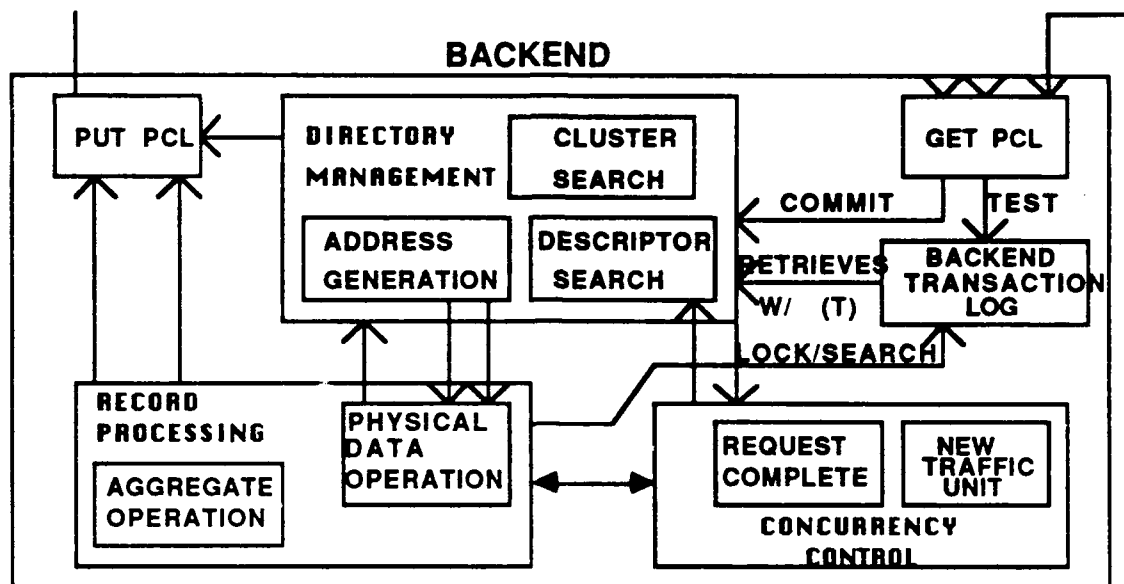


Figure 18. Transaction Inserted into the Database

released. Additionally, the BTL manager releases locks on the BTL request, the transaction is flushed from the BTL, and then the transaction pointer returns to NIL.

If the user selects to UNCOMMIT the transaction, RP again notifies CC to release all locks on the transaction. Additionally, the transaction in the BTL is flushed from the BTL and the transaction pointer returns to NIL.

#### 4. SUMMARY

In this Chapter, I discussed the data structure of the BTL. The data structure of the BTL is an array of pointers where each pointer points to a transaction. The transaction is stored as a linked list of records where each record is a request. Following a discussion of the data structure, we looked at the algorithm from a decomposition point of view. This gave us a "big picture" view of how the algorithm would work. And then finally, we went into the algorithm itself.

The BTL algorithm has many advantages. First, the UNCOMMIT is simple to execute because all that needs to be done is to flush the transaction from the BTL. Second, the COMMIT is done rapidly since all that is required is to send the transaction from the BTL to DIRECTORY\_MANAGEMENT as if the transaction had originated from the controller in the first place. Third, since no modification is made to the database until the user commits, roll-back is done quickly and efficiently. Fourth, since each user has its own test segregated in the BTL and there is

no change to the database, each user is partitioned to the effects of his test only. Fifth, the BTL has a dynamic data structure so there is negligible overhead when no one is in the TEST mode and expands as users enter the TEST mode. And finally, there is no change to the current system software except when the user selects the TEST option. The disadvantages are the extra overhead required when a TEST option has been selected and the intuitive observation that a test RETRIEVE will take longer to execute than a non-test RETRIEVE.



## V. CONCLUSIONS

In Chapter I, I stated that we wanted to design a roll-back and recovery algorithm that took into account nine key factors. The first key factor required that we should develop an algorithm that incurred modest storage overhead for the duplicated data that is brought into main memory. In this algorithm, we accomplished this task by requiring our data structure to only hold data that has not been processed into the database and is therefore not replicated data. So in actuality we have exceeded the goal of modest storage overhead.

The second key factor required that the data structure which stores the data holds only the data it needs and dynamically grow and shrink to fit the system needs. This is exactly how the data structure for the BTL is designed and implemented in the algorithm.

The third key factor required that the roll-back and recovery mechanism is designed with the consideration of storage overhead versus the number of rollbacks required. A system that seldom has a need to roll-back should have a different algorithm than a system that frequently rolls-back. Our system had a unique problem. In the direct COMMIT mode, the system would not roll-back and therefore not require any storage overhead. But in the TEST mode, the system may expect to roll-back many times. This has mandated the requirement not to change the database until the user

committed the transaction. In order to solve this problem in the most efficient manner, we designed a dynamic data structure for the BTL so that in cases when there were no TEST operations ongoing, there was little storage overhead to the system. On the other hand, we created a differential log when in the TEST mode so that we could quickly and efficiently roll-back numerous times.

The fourth key factor required that our design permits parallelism to the maximum extent possible to satisfy system performance requirements. A system could always guarantee a consistent view of the database and fast recovery if it serially executed each transaction. But the costs of serial execution negate any performance benefit of a concurrent system. Additionally, the level of granularity must be to the level that offers maximum concurrency. In order to maintain the concurrency and parallelism of the system, the concurrency control only had to be slightly modified in the TEST mode. The only difference in the TEST mode is that the Record Processing Section (RP) must hold onto its cluster id locks a little longer until it can get an additional lock on the BTL and modify the RETRIEVE REQUEST.

The fifth key factor required that the algorithm performs satisfactorily in a network environment. Incorporating the TEST mode into the system would naturally increase the load to the network because of the increased interaction between the user and the database. However, implementing a

table approach as opposed to modifying the database would appear the least costly of the two alternatives.

The sixth key factor required that the overhead during normal performance should not be increased by the roll-back and recovery mechanism. Since normal performance does not require interface with the BTL, there is no degradation of system performance caused by the roll-back and recovery mechanism.

The seventh key factor required that the recovery speed during roll-back should not cause major delays to the users. Since roll-back only requires flushing the transaction from the BTL and does not require any modification to the database, the roll-back will not create any major delays to the users.

The eighth key factor required that the software complexity of the recovery mechanism needs to be as simple as possible to prevent system delays. Implementing the BTL as an array of pointers pointing to a linked list of records intuitively fits the structure of multiple transactions. The complexity for flushing the BTL requires only the setting of the pointer to the transaction to NIL.

And finally, the recovery system must be reliable. Having not modified the database before a roll-back is initiated allows for a very reliable roll-back

mechanism. Since the state of the database is only updated during COMMIT operations, the database is never in an inconsistent state.

The BTL algorithm meets or exceeds all of the critical factors required of a roll-back and recovery mechanism. It will give MBDS yet another necessary capability.

## APPENDIX

### THE BACK-END TRANSACTION LOG(BTL) ALGORITHM

```
program ROLL-BACK(ouput,input);
(*****)
(* This is an algorithm for implementing a roll-back *)
(* operation that will allow TEST queries with data to be *)
(* entered into the database. Users then will have *)
(* the ability to commit the changes or uncommit them. *)
(*****)

(* DATA STRUCTURE and DECLARATIONS *)

const
    STRINGSIZE = 45;
    BTLSIZE = 25;

type
    LINKPOINTER = ^BTL;
    INDEX = 1..BTLSIZE;
    POINTERARRAY = array [INDEX] of LINKPOINTER;
    STRING = packed array [1..STRINGSIZE] of CHAR;
    BTL = record
        TRANSUM : integer;
        REQNUM : integer;
        ABDLCMD : string;
        COMMIT : boolean;
        TEST : boolean;
        USERNUM : integer;
        FWDPTR : linkpointer;
    end; (* BTL *)

var
    USERREPLY : char;
    TRANSACTION_NUM : integer;
    USER_NUMBER : integer;
    PTRARRAY : pointerarray;
```

```

(*****
(*****  DELETE_FROM_BTL  *****)
(*****
procedure DELETE_FROM_BTL (PTRARRAY(.NUM.) : linkpointer);
(*****
(* This procedure deletes the transaction from the BTL *)
(* and returns the pointer to the system. *)
(*****

```

```

begin
    PTRARRAY(.NUM.) := nil;
end; (* DELETE_FROM_BTL *)

```

```

(*****
(***** ADD_TO_DATABASE *****)
(*****
procedure ADD_TO_DATABASE (PTRARRAY(.NUM.) : linkpointer);
(*****
(* This procedure accesses the transaction in the BTL *)
(* and sends all UPDATES, INSERTs OR DELETEs to *)
(* DIRECTORY_MANAGEMENT for input into the database. *)
(* The "TEST" field is changed to false to tell *)
(* DIRECTORY_MANAGEMENT that this request is not a test *)
(*****

```

var

    TEMPPTR3 : linkpointer;

begin

```

    new (TEMPPTR3);
    TEMPPTR3 := PTRARRAY(.NUM.);
    while TEMPPTR3.FWDPTR <> nil do begin
        read (TEMPPTR3.ABDLCMD);
        if TEMPPTR3.ABDLCMD = "UPDATE, INSERT OR
            DELETE" then begin
            TEMPPTR3.TEST := false;
            DIRECTORY_MANAGEMENT(PTRARRAY(.NUM.),
                TEMPPTR3);
        end (* if *)
        TEMPPTR3 = TEMPPTR3.FWDPTR;
    end; (* while *)
    read (TEMPPTR3.ABDLCMD);
    if TEMPPTR3.ABDLCMD = "UPDATE, INSERT OR DELETE"
        then begin
        TEMPPTR3.TEST := false;
        DIRECTORY_MANAGEMENT(PTRARRAY(.NUM.),TEMP
            PTR3);
    end (* if *)
end; (* ADD_TO_DATABASE *)

```

```

(*****
***** CONTROLLER *****
*****
***** POST_PROCESSING *****
*****)
procedure POST_PROCESSING (RETRIEVE_REQUEST : undefined,

                                PTRARRAY(.NUM.) : linkpointer);
(*****
(* This procedure aggregates all the RETRIEVE_REQUEST(S) *)
(* from the back-ends and then asks the user if *)
(* he/she wants to commit the changes or uncommit them. If the user *)
(* wants to commit the changes then the changes are *)
(* added to the data base (ADD_TO_DATABASE) and *)
(* then deleted from the BTL(DELETE_FROM_BTL). If *)
(* the user wants to uncommit the changes *)
(* then the changes are deleted from the BTL(DELETE_FROM_BTL). *)
*****)

var
    ANSWER : char;

begin
    PP aggregates RETRIEVE_REQUEST (S) from back-ends;
    displayed to user;
    writeln ("COMMIT OR UNCOMMIT, C OR U?");
    readln (ANSWER);
    if ANSWER = "C" then begin
        ADD_TO_DATABASE (PTRARRAY(.NUM.));
        DELETE_FROM_BTL (PTRARRAY(.NUM.));
    end (* if *)
    else
        DELETE_FROM_BTL (PTRARRAY(.NUM.));
end; (* POST_PROCESSING *)

```



```

(*****
(***** SEARCH_BTL2 *****)
(*****
procedure SEARCH_BTL2 (var TEMP_PTR : linkpointer,
                        var FOUND : boolean);
(*****
(* This procedure is called by UPDATE_RETRIEVE to see *)
(* if the request is an INSERT, UPDATE OR DELETE *)
(* which will be used to update the RETRIEVE_REQUEST. *)
(*****

begin
    read (TEMP_PTR.ABDLCMD);
    if TEMP_PTR.ABDLCMD = "INSERT,UPDATE OR DELETE" then
        FOUND := true;
end; (* SEARCH_BTL2 *)

```

```

(*****
(***** UPDATE_RETRIEVE *****)
(*****
procedure UPDATE_PETRIEVE (PTRARRAY(.NUM.) : linkpointer,
                           RETRIEVE_REQUEST : undefined);
(*****
(* This procedure updates the RETRIEVE_REQUEST by *)
(* going through the transaction with another *)
(* pointer and checking for INSERTs,UPDATEs OR DELETES. *)
(* If one is found, then the RETRIEVE_REQUEST is *)
(* modified by record processing. *)
(*****

var
    TEMPPTR2 : linkpointer;
    FOUND : boolean;

begin
    new (TEMPPTR2);
    TEMPPTR2 := PTRARRAY(.NUM.);
    repeat
        FOUND := false;
        SEARCH_BTL2(TEMPPTR2,FOUND);
        if FOUND then begin
            RP compares TEMPPTR2.ABDLCOM to
                RETRIEVE_REQUEST;
            RP modifies RETRIEVE_REQUEST;
            TEMPPTR2 := TEMPPTR2.FWDPTR;
        end; (* if *)
    until TEMPPTR2.FWDPTR = nil
    FOUND := false;
    SEARCH_BTL2(TEMPPTR2, FOUND);
    RP compares TEMPPTR2.ABDLCOM to RETRIEVE_REQUEST;
    RP modifies RETRIEVE_REQUEST;
    POST_PROCESSING (RETRIEVE_REQUEST,
        PTRARRAY(.NUM.));
end; (* UPDATE_RETRIEVE *)

```

```

(*****)
(***** DIRECTORY MANAGEMENT *****)
(*****)
procedure DIRECTORY_MANAGEMENT(PTRARRAY(.NUM.)
                                linkpointer,
                                TEMPPTR : linkpointer);
(*****)
(* This procedure takes in any ABDL request and *)
(* processes the query. If the query is a test query *)
(* then it has to request an additional lock for the BTL. *)
(* It also calls UPDATE_RETRIEVE to recompute *)
(* the RETRIEVE_REQUEST which will eventually be shown to the user. *)
(*****)

var
  DM (* DIRECTORY MANAGEMENT *) : string;
  CC (* CONCURRENCY CONTROL *) : string;
  RP (* RECORD PROCESSING *) : string;
  BTL (* BACK-END TRANSACTION LOG *) : string;
  PP (* POST PROCESSING *) : string;

begin
  DM sends type-C attributes needed from TEMPPTR to CC
    - when attributes locked then
      + CC signals DM;

  DM performs descriptor search
    - signals CC to release locks on those attributes;

  DM sends the descriptor-ID groups for retrieve request to CC
    - when descriptor-ID groups locked and cluste. search allowed
      then
        + CC signals DM;

  DM performs cluster search and signals CC to release the locks on the
    descriptor-ID groups;

```

```

DM sends the cluster-IDs for the retrieval to CC.
  - once cluster-IDs are locked
    + address generation
    + if TEMP_PTR.ABDLCMD = RETRIEVE OR
      RETRIEVE-COMMON then
      + begin
        -request processed
        -RETRIEVE_REQUEST formed
      + end (* if *)
  - CC signals DM;

if TEMP_PTR.TEST = true then begin
  RP requests lock on BTL to CC
  when lock is approved
    UPDATE_RETRIEVE (PTRARRAY(.NUM.),
                     RETRIEVE_REQUEST);
    RP returns locks on cluster-ids to CC
    RP returns locks on BTL
  end; (* if *)
end; (* DIRECTORY_MANAGEMENT *)

```

```

(*****
(***** SEARCH_BTL *****)
(*****
procedure SEARCH_BTL (PTRARRAY (.NUM.) : linkpointer);
(*****
(* This procedure searches the transaction for retrieves *)
(* and sends the retrieve to DIRECTORY MANAGEMENT for *)
(* processing. *)
(*****

var
    TEMPPTR : linkpointer;

begin
    TEMPPTR := PTRARRAY(.NUM.);
    while TEMPPTR.FWDPTR <> nil do begin
        read (TEMPPTR.ABDLCMD);
        if TEMPPTR.ABDLCMD = "RETRIEVE" then
            DIRECTORY_MANAGEMENT(PTRARRAY(.NUM.),
                                TEMPPTR);
        TEMPPTR = TEMPPTR.FWDPTR;
    end; (* while *)
    if TEMPPTR.ABDLCMD = "RETRIEVE" then
        DIRECTORY_MANAGEMENT(PTRARRAY(.NUM.),
                            TEMPPTR);
end; (* SEARCH_BTL *)

```

```

(*****
(***** INSERT_TRANS *****)
(*****
procedure INSERT_TRANS (var PTRARRAY(.NUM.) : linkpointer,
                        BTL : record);
(*****
(* This procedure takes the record (request) from *)
(* INITIALIZE and joins it to the rest of the transaction *)
(* in the BTL. *)
(*****

var
    TEMPPTR : linkpointer;

begin
    new(TEMPPTR);
    if PTRARRAY(.NUM.) = nil then begin
        PTRARRAY(.NUM.) = BTL;
        PTRARRAY(.NUM.).FWDPTR = nil;
    end
    else begin
        TEMPPTR = PTRARRAY(.NUM.);
        while TEMPPTR.FWDPTR <> nil do
            TEMPPTR = TEMPPTR.FWDPTR;
        end;
        TEMPPTR.FWDPTR = BTL;
        BTL.FWD = nil;
    end;
end; (* INSERT_TRANS *)

```

```

(*****
(***** INITIALIZE *****)
(*****
procedure INITIALIZE (var PTRARRAY(.NUM.) : linkpointer);
(*****
(* This procedure inserts the complete transaction into *)
(* the data structure (BTL). It takes one request at *)
(* a time, puts the request into a record, and then *)
(* sends the record to INSERT_TRANS which attaches *)
(* the record to the transaction in the BTL. *)
(*****

```

var

```

    ENTERREQUEST : boolean;
    ANSWER : char;
    REQUESTNUM : integer;
    USER_NUM : integer;

```

begin

```

    writeln ("ENTER YOUR USER NUMBER");
    readln (USER_NUM);
    REQUESTNUM := 0;
    BTL.TRANSUM := NUM;
    ENTERREQUEST := true;
    writeln ("TYPE 'BEGIN'");
    while ENTERREQUEST = true do begin
        REQUESTNUM := REQUESTNUM + 1;
        writeln ("ENTER YOUR REQUEST");
        readln (BTL.ABDLCMD);
        BTL.REQNUM = REQUESTNUM;
        BTL.PARTIALCOM = true;
        BTL.COMMIT = false;
        BTL.TEST = true;
        BTL.USERNUM = USER_NUM;
        INSERT_TRANS (PTRARRAY(.NUM.),BTL);
        WRITELN ("FINISHED ENTERING REQUESTS? Y OR N");
        READLN (ANSWER);
        if ANSWER = 'N' then
            ENTERREQUEST = true
        else
            ENTERREQUEST = false
    end; (* while ENTERREQUEST *)
    writeln ("TYPE 'END'");
end; (* INITIALIZE *)

```

```

(*****)
(***** SEARCH_FOR_PTRARRAY *****)
(*****)
procedure SEARCH_FOR_PTRARRAY(var PTRARRAY(.NUM.) :
                               linkpointer);
(*****)
(* This procedure searches a global array of pointers *)
(* for an empty pointer to insert the transaction into *)
(* the Back-end Transaction Log. *)
(*****)

var
    MAXLENGTH, COUNT : index;
    STOPLOOP : boolean;

begin
    STOPLOOP := false;
    for COUNT = 1 to MAXLENGTH do begin
        if STOPLOOP := false then begin
            if PTRARRAY(.NUM.).FWDLINK = nil then begin
                STOPLOOP := true;
            end; (* if *)
            NUM := NUM + 1;
        end; (* if *)
    end; (* for *)
    NUM := NUM - 1;
end; (* SEARCH_FOR_PTRARRAY *)

```



```

(*****
(***** TEST *****)
(*****
procedure TEST;
(*****
(* This procedure is the "junction" of this algorithm. *)
(* It calls SEARCH_FOR_PTRARRAY to find a slot in the BTL. *)
(* It then calls INITIALIZE to insert the *)
(* transaction into the BTL. And then calls SEARCH_BTL *)
(* to run the transaction. *)
(*****

```

```

var
    NUM : index;

begin
    NUM := 1;
    SEARCH_FOR_PTRARRAY(PTRARRAY(.NUM.));
    INITIALIZE (PTRARRAY(.NUM.));
    SEARCH_BTL (PTRARRAY(.NUM.));
end;(* TEST *)

```

```

(*****
(***** CREATE_PTR_ARRAY *****)
(*****
procedure CREATE_PTR_ARRAY;
(*****
(* This procedure creates the array that stores the BTL. *)
(*****

```

```

var
    I, STOP = index;

begin
    for I = 1 to STOP do begin
        PTRARRAY(I.) = nil;
    end; (* for *)
end; (* CREATE_PTR_ARRAY *)

```

```

(*****
(***** M A I N *****
(*****
begin
    writeln ("Type C for commit, T for test, D for data dictionary");
    writeln ("or Type Q for quit");
    CREATE_PTR_ARRAY;
    readln (USERREPLY);
    while USERREPLY in [.C,c,T,t,D,d.] do begin
        case USERREPLY of
            C,c : COMMIT_PROCEDURE;
            T,t : TEST;
            D,d : DATA_DICTIONARY_PROCEDURE;
        end (* while case *)
        writeln ("Type C for commit, T for test, D for data");
        writeln ("dictionary");
        writeln ("or Type Q for quit");
        readln (USERREPLY);
    end; (* while USERREPLY *)

end. (* ROLL-BACK *)

```

## LIST OF REFERENCES

1. Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems," *ACM Transactions on Database Systems*, v. 5, pp. 139-156, June 1980.
2. Kohler, W., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Transactions on Database Systems*, v. 13, pp. 149-179, June 1981.
3. Gray, J., and others, "The Recovery Manager of the System R Database Manager," *ACM Transactions on Database Systems*, v. 13, pp. 223-242, June 1981.
4. Aghili, H., and Severance, D., "A Practical Guide to the Design of Differential Files for Recovery of On-Line Databases," *ACM Transactions on Database Systems*, v. 7, pp. 540-565, December 1982.
5. Cardenas, A., Alavian, F., and Avizienis, A., "Performance of Recovery Architectures in Parallel Associative Database Processor," *ACM Transactions on Database Systems*, v. 8, pp. 291-323, September 1983.
6. Hecht, M., and Gabbe, J., "Shadowed Management of Free Disk Pages with a Linked List," *ACM Transactions on Database Systems*, v. 8, pp. 503-514, December 1983.
7. Reuter, A., "Performance Analysis of Recovery Techniques," *ACM Transactions on Database Systems*, v. 9, pp. 526-559, December 1984.
8. Bernstein, P., and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems*, v. 9, pp. 596-615, December 1984.
9. Agrawal, R., and Dewitt, D., "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," *ACM Transactions on Database Systems*, v. 10, pp. 529-564, December 1985.

10. IBM Research Division, Report RJ 6649, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, by C. Mohan and others, pp. 1-45, 23 January 1989.
11. IBM Research Division, Report RJ 6650, *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*, by K. Rothermel and C. Mohan, pp. 1-22, 23 January 1989.
12. The Ohio State University, Columbus, Ohio, Report OSU-CISRC-TR-81-7, *"Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion, and Capacity Growth (Part 1)"*, by D.K. Hsiao and M.J. Menon, August 1981.
13. The Ohio State University, Columbus, Ohio, Report OSU-CISRC-TR-81-8, *"Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion, and Capacity Growth (Part 2)"*, by D.K. Hsiao and M.J. Menon, August 1981.
14. Naval Postgraduate School, Report NPS52-83-003, *"The Implementation of a Multi-Backend Database System (MDBS): Part III - The Message-Oriented Version With Concurrency Control and Secondary-Memory-Based Directory Management,"* by Hsiao, D.K., Boyne, R.D., and Demurjian, S.A., pp. 1-89, March 1983.
15. Naval Postgraduate School, Report NPS52-84-005, *"The Implementation of a Multi-Backend Database System (MDBS): Part IV - The Revised Concurrency Control and Directory Management Processes and the Revised Definitions of Inter-process and Inter-computer Messages,"* by Hsiao, D.K., Kerr, D.S., and Demurjian, S.A., pp. 1-121, February 1984.
16. Hsiao, D.K., and Demurjian, S.A., "Towards a Better Understanding of Data Models Through the Multilingual Database System," *IEEE Transactions on Software Engineering*, v. 14, pp. 946-958, July 1988.
17. Rodeck, B.D., *Accessing and Updating Functional Databases Using Codasyl-DML*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1986.
18. Korth, H.F., Silberschatz, A., *Database System Concepts*, McGraw-Hill, 1986.

## BIBLIOGRAPHY

- Benson, T.P., and Wentz, G.L., *The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.
- Bernstein, P.A., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- Hsiao, D.K., *Modern Database System Architectures*, (Unpublished Manuscript), March 1989.
- Hsiao, D.K., Coker, H., and Demurjian, S.A., *The Multi-Model Database System*, Naval Postgraduate School, Report NPS52-87-026 June 1987.
- Hsiao, D.K., and Kamel, M.N., *Heterogeneous Databases: Proliferations, Issues and Solutions*, IEEE Transactions on Knowledge and Data Engineering, KDE 1, 1 (June 1989).
- Hsiao, D.K., Pitargue, M., and Wong, A., *Reliable Broadcasting for Parallel Database Backend Computers*, Naval Postgraduate School, Report NPS52-89-001, November 1988.
- Zawis, J.A., *Accessing Hierarchical Databases via SQL Transactions in a Multi-Model Database System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1987.

## INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5100	1
5.	Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
6.	CPT David E. Quantock 883 Jefferson Dr. Greenville, MS 38701	2
7.	Marciano Pitargue, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	1
8.	Debbie Gaiser, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	1
9.	Tom Chu, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	1

- |     |   |   |
|-----|---|---|
| 10. | Steven A. Demurjian, Code 52<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, California 93943-5100 | 1 |
| 11. | Timothy P. Benson<br>P.O. Box 1974<br>Woodbridge, Virginia 22193  | 1 |
| 12. | Gary L. Wentz<br>111 Appian Way<br>Pasadena, Maryland 21122   | 1 |
| 13. | Gary R. Kloepping<br>Route 1, Box 99<br>Santa Rosa, Texas 78593   | 1 |
| 14. | John F. Mack<br>2934 Emory Street<br>Columbus, Georgia 31903  | 1 |